Software libre

David Aycart Pérez
Marc Gibert Ginestà
Martín Hernández Matías
Jordi Mas Hernández

XP06/M2112/01486

Ingeniería del software en entornos de SL



Jordi Mas Hernández

Coordinador

Ingeniero de software en la empresa de código abierto Ximian, donde trabaja en la implementación del proyecto libre Mono. Como voluntario, colabora en el desarrollo del procesador de textos Abiword y en la ingeniería de las versiones en catalán del proyecto Mozilla y Gnome. Coordinador general de Softcatalà. Como consultor ha trabajado para empresas como Menta, Telépolis, Vodafone, Lotus, eresMas, Amena y Terra España.

David Megías Jiménez

Coordinador

Ingeniero de Informática por la UAB. Magíster en Técnicas Avanzadas de Automatización de Procesos por la UAB. Doctor en Informática por la UAB.

Profesor de los Estudios de Informática y Multimedia de la UOC.

David Aycart Pérez

Autor

Director de desarrollo formativo de Free Software Certification.

Socio fundador de Free Software Certification.

Socio fundador de Esware Linux S.A.

Marc Gibert Ginestà

Autor

Ingeniero en Informática por la Universidad Ramon Llull. Socio fundador y jefe de proyectos de Cometa Technologies, empresa dedicada a dar soluciones en tecnologías de la información, basadas en el uso de estándares y herramientas de código abierto. Profesor del Máster de Seguridad en Tecnologías de la Información en Ingeniería y Arquitectura La Salle y consultor del Máster Internacional de Software Libre de la UOC (Universitat Oberta de Catalunya).

Martín Hernández Matías

Autor

Ingeniero técnico informático de la Universidad Politécnica de Madrid.

Training manager de Free Software Certification.

Segunda edición: febrero 2007

Av. Tibidabo, 39-43, 08035 Barcelona
Material realizado por Eureca Media, SL

Depósito legal: B-12.659-2007

Índice

ANOTACIONES

ANOTACIONES

ANOTACIONES

ANOTACIONES

Agradecimientos

Los autores agradecen a la Fundación para la Universitat Oberta de Catalunya (http://www.uoc.edu) la financiación de la primera edición de esta obra, enmarcada en el Máster Internacional en Software Libre ofrecido por la citada institución.

ANOTACIONES

1. Introducción a la ingeniería del software

1.1. Introducción

Este primer capítulo del curso *Ingeniería del software en entornos de software libre*, proporciona las bases para conocer los conceptos principales involucrados en la ingeniería del software en general, y su aplicación al software libre en particular.

A lo largo del capítulo, veremos que la ingeniería del software es casi tan antigua como el propio software, y también que, parte de los problemas que hicieron nacer este sector de la industria del software siguen vigentes, la mayoría sin una solución clara y definida. Pese a ello, se han hecho grandes avances en todos los campos, desde la estimación del coste, pasando por la gestión de los equipos de trabajo, documentación, y muy notablemente en pruebas, calidad y seguridad.

Después de introducir brevemente los conceptos más importantes, pasamos directamente a examinar dos metodologías concretas bastante populares. En primer lugar, veremos la metodología eXtreme *Programming*, quizá la más próxima a la forma de organizar y participar en proyectos de software libre. A continuación, daremos un vistazo a Métrica v3, una metodología clásica que es imprescindible conocer si se han de realizar proyectos con la Administración española.

Existen cientos de metodologías (públicas y particulares de empresas), pero los conceptos introducidos a lo largo de este módulo nos proporcionarán los conocimientos necesarios para evaluar su idoneidad en nuestro ámbito de trabajo o proyecto concreto, y para profundizar en los aspectos de la ingeniería del software que más nos hayan llamado la atención.

ANOTACIONES

1.2. Objetivos

- Tomar conciencia de la importancia de los procesos y actividades relacionados con la ingeniería del software en proyectos de todo tipo.
- Conocer los conceptos básicos involucrados, que nos permitirán documentarnos más ampliamente cuando nos sea necesario, y evaluar metodologías concretas.
- Familiarizarse con la metodología eXtreme Programming, y con los nuevos paradigmas de gestión, diseño y desarrollo que plantea.
- Conocer la estructura general de la metodología Métrica v3, y sus principales procesos, actividades e interfaces.

1.3. Un poco de historia

El término *ingeniería del software* empezó a usarse a finales de la década de los sesenta, para expresar el área de conocimiento que se estaba desarrollando en torno a las problemáticas que ofrecía el software en ese momento.

En esa época, el crecimiento espectacular de la demanda de sistemas de computación cada vez más y más complejos, asociado a la inmadurez del propio sector informático (totalmente ligado al electrónico) y a la falta de métodos y recursos, provocó lo que se llamó *la crisis del software* (en palabras de Edsger Dijkstra) entre los años 1965 y 1985.

Durante esa época muchos proyectos importantes superaban con creces los presupuestos y fechas estimados, algunos de ellos eran tan críticos (sistemas de control de aeropuertos, equipos para medicina, etc.) que sus implicaciones iban más allá de las pérdidas millonarias que causaban.

La crisis del software pasó, no tanto por la mejora en la gestión de los proyectos, sino en parte porque no es razonable estar en crisis

Nota

Alguno de los proyectos más representativos de la época, como el desarrollo del sistema OS/360 de IBM, tardó más de una década en finalizarse, y fue el primero que involucró a más de 1.000 programadores. Más adelante, el jefe del proyecto en "Mythical Man Month", Fred Books, reconocería que se habían cometido errores de millones de dólares, y pronunciaría la conocida ley de Brooks:

Asignar más programadores a un proyecto ya retrasado, suele retrasar aún más el proyecto.

ANOTACIONES

más de veinte años, y en parte porque se estaban haciendo progresos en los procesos de diseño y metodologías.

Así pues, desde 1985 hasta el presente, han ido apareciendo herramientas, metodologías y tecnologías que se presentaban como la solución definitiva al problema de la planificación, previsión de costes y aseguramiento de la calidad en el desarrollo de software.

Entre las herramientas, la programación estructurada, la programación orientada a objetos, a los aspectos, las herramientas CASE, el lenguaje de programación ADA, la documentación, los estándares, CORBA, los servicios web y el lenguaje UML (entre otros) fueron todos anunciados en su momento como la solución a los problemas de la ingeniería del software, la llamada "bala de plata" (por *silver bullet*). Y lo que es más, cada año surgen nuevas ideas e iniciativas encaminadas a ello.

Por supuesto, también ha habido quien ha culpado a los programadores por su indisciplina o anarquía en sus desarrollos. La ignorancia y algunos casos excéntricos contribuyeron a crear una imagen falsa del programador, que hoy en día aún perdura. Aunque muchas veces él es el "sufridor" de alguna de estas metodologías o de una pobre implementación de las mismas, parece lógico que, como participante activo en el proyecto, las metodologías más modernas empiecen a tenerle más en cuenta.

En combinación con las herramientas, también se han hecho esfuerzos por incorporar los métodos formales al desarrollo de software, argumentando que si se probaba formalmente que los desarrollos hacían lo que se les requería, la industria del software sería tan predecible como lo son otras ramas de la ingeniería.

Entre las metodologías y procesos, además de Métrica v3 (promovida por la Secretaría del Consejo Superior de Informática) y eXtreme Programming, que veremos en detalle más adelante, destacan muchos otros como RUP (*rational unified process* desarrollado por Rational Software Corp. ahora una división de IBM), SSADM (*structured systems analysis and design methodology* promovido por el Gobierno británico) o el método de evaluación de la capacidad de desarrollo de los equipos o empresas conocido como CMMI (*capability maturity model*

ANOTACIONES

integration). Paralelamente, suelen usarse también métodos de predicción de costes como COCOMO o los puntos de función.

Las últimas iniciativas en este campo son múltiples y se extienden a lo largo de todo el proceso relacionado con el software. Los más académicos se inclinan por una estructura de componentes, servicios, con orientación a objetos o a aspectos en la implementación; aunque también es igual de significativo el desarrollo de las herramientas que nos ayuden a representar y compartir estos diseños, así como a valorar el esfuerzo y el valor que añaden al producto final. Es realmente un campo fascinante, donde tanto en los seminarios o en las grandes consultoras, como en los pequeños laboratorios se innova cada día y se presentan buenas ideas.

1.4. Conceptos

En este capítulo, veremos los conceptos más comunes asociados con la ingeniería del software, aplicables tanto a entornos de software libre, como al resto. Todos ellos disponen de abundante bibliografía, y de algunos cursos y másters especializados en su aprendizaje y aplicación a los procesos de desarrollo. Aquí únicamente los definiremos para tener una base sobre la que tratar los ejemplos de metodología que veremos en los capítulos siguientes.

1.4.1. Gestión de proyectos

La gestión de proyectos es la disciplina que agrupa y ordena el conjunto de tareas o actividades destinadas a obtener unos objetivos. Esto incluye la planificación, definición, ordenamiento y gestión de las actividades que formarán el proyecto software.

En su primera expresión, la buena gestión de un proyecto es la que es capaz de reducir al mínimo las posibilidades de fallo de éste durante todo el transcurso del mismo. Pudiendo entenderse como fallo, la no consecución de los requisitos iniciales del proyecto, la inviabilidad económica del mismo, un resultado que impida mantenerlo o le impida evolucionar, etc.

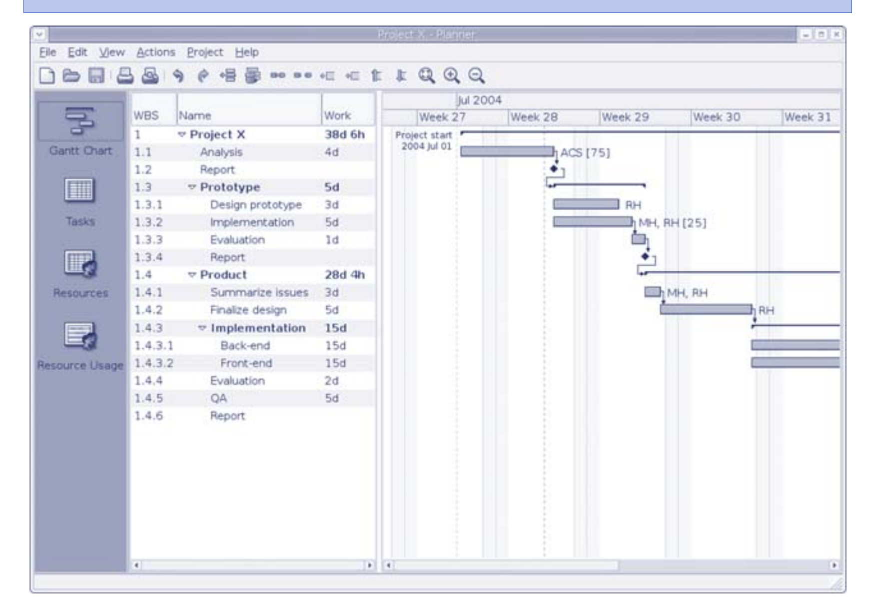
ANOTACIONES

Otra visión más orientada al ámbito de costes y económico del proyecto es la que define la buena gestión como la que optimiza el uso de los recursos (tiempo, dinero, personas, equipos, etc.) en cada fase del mismo.

Muy comúnmente, la gestión del proyecto se lleva a cabo por el director del proyecto, normalmente una sola persona que no tiene por qué participar activamente en las actividades del mismo, pero que sí que se ocupa de monitorizar su progreso y de la interacción entre los diferentes grupos que intervienen en éste para minimizar el riesgo de fallo del proyecto.

Además de hojas de cálculo de control de horas por tarea, uso de recursos, etc., los diagramas de *Gantt* son muy usados en estos entornos por mostrar de una forma clara la sucesión de tareas, recursos involucrados y sus dependencias.

Figura 1. Captura de pantalla del software GPL Imendio Planner mostrando un diagrama de Gantt



En los diagramas de *Gantt*, representamos las actividades que hay que realizar en forma de árbol (mitad izquierda de la figura), indicando su fecha de inicio y su duración estimada. El programa representa la actividad sobre un calendario (mitad derecha de la figura) y nos permite definir las dependencias entre las actividades. Al indicar que una tarea debe empezarse al terminar otra, o que deben empezar a la vez o terminar las dos simultáneamente, el diagrama va modificando automáticamente la situación de las actividades en el tiempo.

ANOTACIONES

Nota

El camino crítico de un proyecto se define como la secuencia de tareas que suman el intervalo de tiempo más largo del mismo. Así pues, determina la duración mínima del proyecto, ya que un retraso en alguna de sus tareas tiene un impacto directo en su duración total.

Dependiendo de la complejidad del programa que utilicemos para representar el diagrama, podremos especificar también los recursos disponibles, los necesarios para cada tarea, etc. y seremos capaces de detectar conflictos en la planificación, como intervalos en los que no dispondremos de los recursos necesarios, o cuál va a ser el camino crítico del proyecto (la sucesión de tareas que por sus dependencias van a determinar la duración máxima del proyecto).

Concretamente, en entornos de software libre, no existe una forma "tradicional" de gestionar un proyecto. Un buen ejemplo de ello es el artículo de Eric S. Raymond, "La Catedral y el Bazar", en el que se describe la forma tradicional de gestionar un proyecto como analogía de construir una catedral, y se compara la forma como se gestionan muchos proyectos de software libre al funcionamiento de un bazar.

Proyectos grandes de software libre, como el desarrollo del núcleo Linux o el servidor web Apache, están formados por un comité de gestión del proyecto (o una sola persona como dictador benevolente) que deciden los siguientes pasos que habrá que llevar a cabo en el proyecto, y que inmediatamente delegan en otras personas o equipos los cambios que hay que realizar para llegar a los objetivos. La aprobación de los cambios realizados o bien la incorporación de nuevas funcionalidades propuestas por los usuarios u otros desarrolladores sigue el proceso inverso hasta llegar al comité de aprobación que decide si se incorporan al proyecto o no.

Web

http://www.bugzilla.org/
http://sourceforge.net/

Por supuesto, en este tipo de gestión de proyectos, juega un papel fundamental las herramientas de soporte y comunicación entre los diferentes equipos. Programas de gestión de incidencias como el conocido *Bugzilla* o entornos como *Sourceforge* que ayudan a los gestores del proyecto o a los comités organizadores a conocer el estado del proyecto, la opinión de sus usuarios, las aportaciones de los desarrolladores, etc.

Proyectos pequeños o medianos de software libre no acostumbran a seguir una metodología tradicional, y se orientan más a una gestión ágil del proyecto, siguiendo principios acordes con el software libre, primando la publicación de nuevas funcionalidades y versiones del proyecto al cumplimiento de un calendario rígido de actividades e hitos.

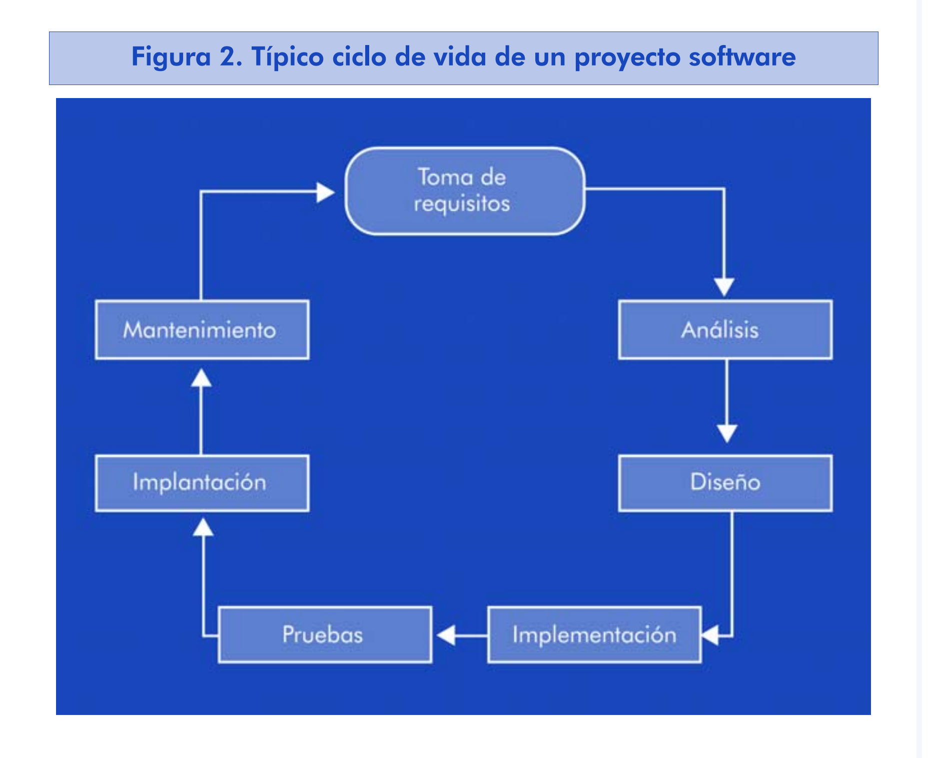
En estos proyectos, el director del mismo suele intervenir muy estrechamente en el desarrollo y en el resto de actividades.

Este último tipo de proyectos suelen gestionarse mediante simples listas de requisitos y fallos por resolver, que por la naturaleza distribuida de los proyectos de software libre, deberá estar disponible para su consulta y modificación por parte de los desarrolladores y usuarios. El gestor o los desarrolladores serán los que den prioridad a las tareas, y decidan cuáles van a incorporar en las nuevas *releases* del proyecto.

1.4.2. Ciclo de vida del software

Se llama ciclo de vida del software a las fases por las que pasa un proyecto software desde que es concebido, hasta que está listo para usarse. Típicamente, incluye las siguientes actividades: toma de requisitos, análisis, diseño, desarrollo, pruebas (validación, aseguramiento de la calidad), instalación (implantación), uso, mantenimiento y obsolescencia.

Figura 2. Típico ciclo de vida de un proyecto software



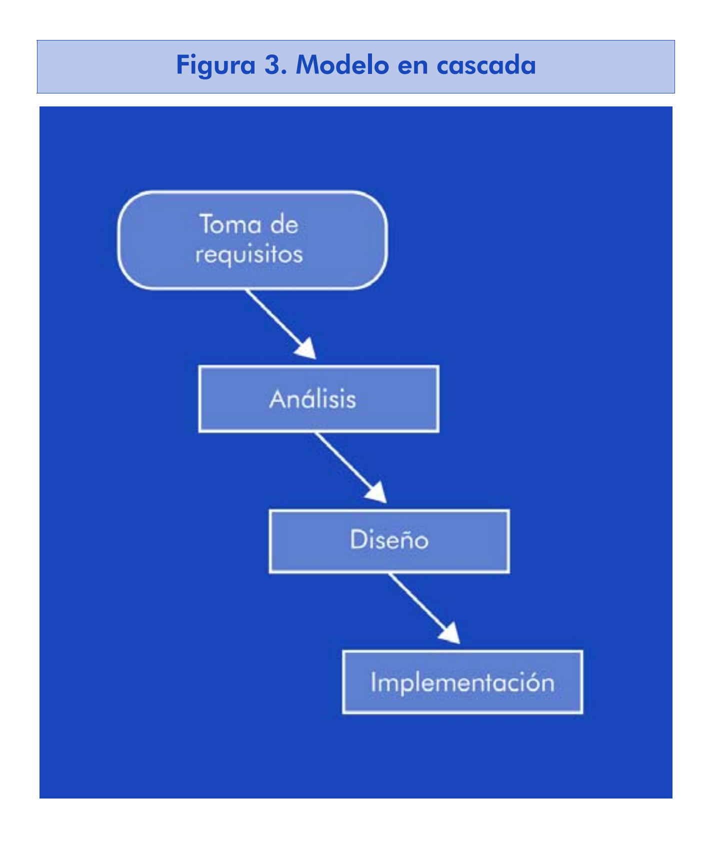
El proyecto tiende a pasar iterativamente por estas fases, en lugar de hacerlo de forma lineal. Así pues, se han propuesto varios modelos (en cascada, incremental, evolutivo, en espiral, o concurrente, por citar algunos) para describir el progreso real del proyecto.

ANOTACIONES

El modelo en cascada es el más simple de todos ellos y sirve de base para el resto. Simplemente asigna unas actividades a cada fase, que servirán para completarla y para proporcionar los requisitos de la siguiente. Así, el proyecto no se diseña hasta que ha sido analizado, o se desarrolla hasta que ha sido diseñado, o se prueba hasta que ha sido desarrollado, etc.

Los modelos incremental y evolutivo son una variación del modelo en cascada en la que éste se aplica a subconjuntos del proyecto. Dependiendo de si los subconjuntos son partes del total (modelo incremental) o bien versiones completas pero con menos prestaciones (modelo evolutivo) estaremos aplicando uno u otro.

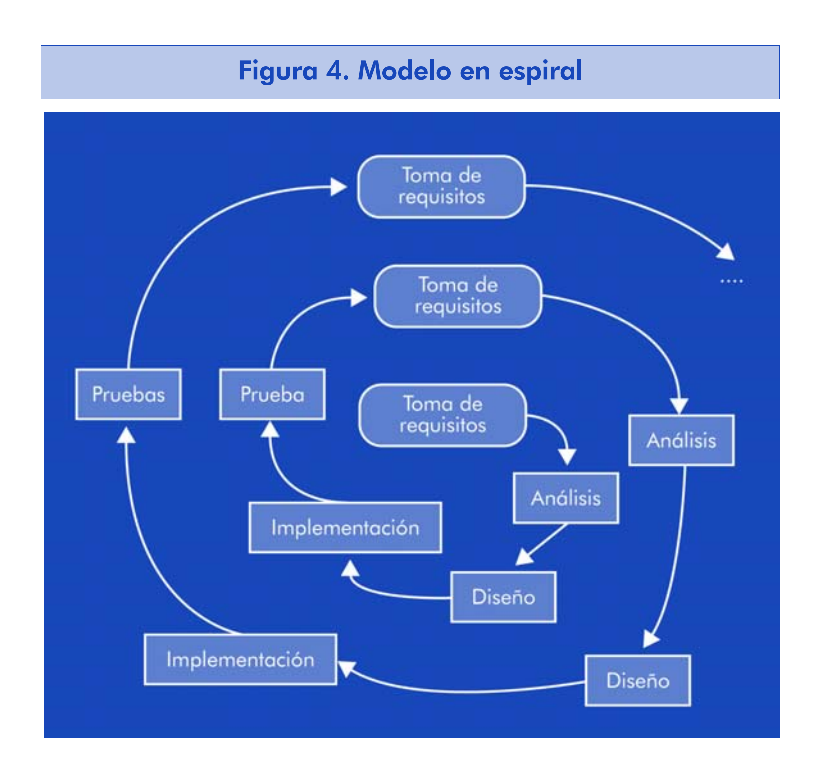
Figura 3. Modelo en cascada



El modelo en espiral se basa en la creación de prototipos del proyecto, que pasan por las fases anteriores, y que van acercándose sucesivamente a los objetivos finales. Así pues, nos permite examinar y validar repetidamente los requisitos y diseños del proyecto antes de acometer nuevas fases de desarrollo.

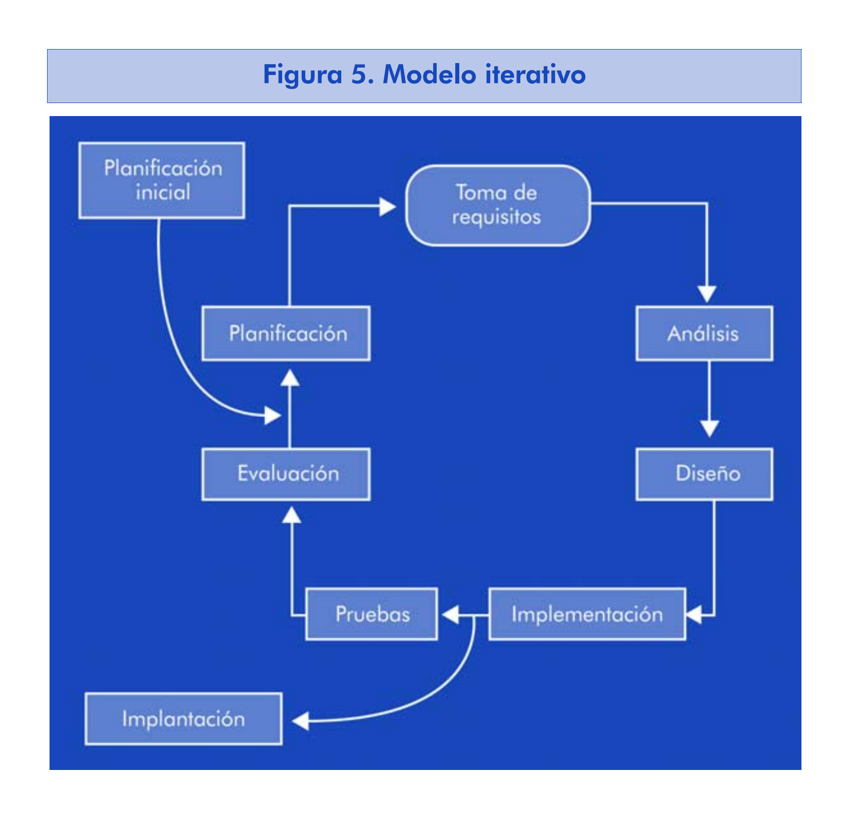
ANOTACIONES

Figura 4. Modelo en espiral



Finalmente, el modelo iterativo o incremental es el que permite que las fases de análisis, diseño, desarrollo y pruebas se retroalimenten continuamente, y que empiecen lo antes posible. Permitirá atender a posibles cambios en las necesidades del usuario o a nuevas herramientas o componentes que los desarrolladores descubran y que faciliten el diseño o proporcionen nuevas funcionalidades.

Figura 5. Modelo iterativo



ANOTACIONES

Se trata de obtener lo más rápidamente una versión funcional del software, y añadirle prestaciones a partir de lo que se ha aprendido en la versión anterior. El aprendizaje proviene tanto del desarrollo anterior, como del uso del software, si es posible. En este tipo de desarrollo es imprescindible establecer una lista de control del proyecto, donde iremos registrando las funcionalidades que faltan por implementar, las reacciones de los usuarios, etc. y que nos proporcionará las bases para cada nueva iteración.

Este último método es el más usado (aunque sea involuntariamente) en proyectos de software libre, por la propia naturaleza cambiante de los requisitos o la aportación constante de nuevos colaboradores.

1.4.3. Análisis de requisitos

El análisis de requisitos es la primera fase de la vida de un proyecto. En ella, habrá que recoger tanto las necesidades del usuario del producto, al más alto nivel, como la especificación de los requisitos software del sistema.

Para guiarnos en ese proceso, existen multitud de plantillas, libros y metodologías orientados a extraer del usuario (o cliente) lo que realmente quiere que haga el sistema. Sin duda, no es una tarea sencilla. El proyecto ReadySET hospedado en http://www.tigris.org proporciona un conjunto de plantillas de documentos para la gestión de un proyecto de desarrollo software, extremadamente útil y de código abierto.

Siguiendo estas plantillas, vemos que la especificación de requisitos del sistema incluye, como mínimo:

- Los casos de uso: actores que intervendrán en el uso del producto y sus posibles acciones.
- Sus requisitos funcionales: prestaciones del producto en la primera versión y posible planificación para las futuras versiones.
- Sus requisitos no funcionales: de rendimiento, de usabilidad, de seguridad, etc.

ANOTACIONES

En entornos de software libre, muy frecuentemente el fundador (y gestor) del proyecto será también su primer usuario, pero esto no le exime de esta tarea, que determinará el alcance del proyecto al menos en sus primeras versiones, y permitirá a los desarrolladores que se vayan incorporando al proyecto, hacerse una idea de su progreso y objetivos, así como dónde prestar su ayuda.

1.4.4. Estimación de costes

La estimación de costes (recursos, equipos y tiempo empleado) es una de las razones de ser de la ingeniería del software. Aunque no siempre aplicable en entornos de software libre, donde no suele ser posible realizar una planificación de recursos disponibles, es conveniente conocer las métricas y métodos que nos permitirán predecir el esfuerzo que supondrá implementar un sistema o alguna de sus prestaciones.

La estimación suele realizarse basándose en modelos matemáticos que parten del "tamaño" estimado del proyecto, y de constantes que lo ajustan según las tecnologías usadas, recursos de que disponemos, etc. Los modelos nos permiten estimar el esfuerzo requerido (habitualmente en horas/hombre o meses/hombre) para terminar el proyecto.

Obviamente, la clave está en estimar el "tamaño" del proyecto. Aun sabiendo lo poco indicativo que puede llegar a ser, son bastantes los modelos que usan las líneas de código para determinar el tamaño de un proyecto (COCOMO, COCOMO II). Otros modelos usan los denominados "puntos de función", que son una medida de la complejidad de cada funcionalidad a partir de las entradas que recibe, las salidas que aporta, su interacción con otras funcionalidades o recursos, etc.

Existen también métricas para lenguajes orientados a objetos, que tienen en cuenta factores como los métodos por clase, las herencias entre objetos, la interacción entre ellos, su grado de dependencia (que complica el mantenimiento, etc.).

No es el objetivo de este capítulo ahondar en estos conceptos, que requerirían muchas páginas de fórmulas, tablas, así como casos de

ANOTACIONES

éxito o fracaso en la estimación. El estudiante interesado en profundizar encontrará mucha información sobre ellos en la bibliografía.

1.4.5. Diseño

La fase de diseño del sistema producirá un conjunto de documentos que describirán un sistema que cumpla con los objetivos de la fase de análisis de requisitos. Las decisiones que se tomen en esta fase deberán basarse en esos requisitos y en la comprensión de la tecnología y los componentes disponibles.

Debe tenerse en cuenta que muchos de los documentos que se crearán en esta fase van a usarse por los desarrolladores del proyecto (diagramas de componentes del sistema, arquitectura del mismo, etc.). Esto implica que deben hacerse lo más completos posible, y que la participación de los desarrolladores en esta fase ayudará a evitar revisiones innecesarias.

Otros documentos irán destinados a los usuarios del proyecto (p. ej., el diseño de la interfaz de usuario), y su aprobación y entendimiento de éstos también será clave para evitar desviaciones.

Por todo ello, es muy conveniente disponer de listas que nos permitan comprobar que no hemos olvidado ningún aspecto clave en esta fase. El proyecto ReadySET dispone de plantillas de casi todos los documentos que podemos necesitar, con los cuestionarios correspondientes y ejemplos.

Extrayendo lo más significativo, encontramos:

- Diagrama estructural: diseño y notas sobre la estructura del sistema en detalle.

- Diagrama de comportamiento: diseño y notas sobre el comportamiento del sistema.

- Arquitectura del sistema: diseño de sus componentes, de su implantación e integración.

ANOTACIONES

- Organización de código fuente y compilación: Directorios, opciones de compilación, sistemas de control de versiones, etc.

- Interfaz de usuario: metáforas, estándares a seguir, diseño de los contextos de interacción con el usuario, etc.

- Sistema de información: bases de datos, abstracción de objetos, almacenamiento, persistencia, etc.

- Seguridad.

1.4.6. Documentación

La documentación de un proyecto es de vital importancia para su éxito. Ya desde la fase de diseño, como parte de la propia arquitectura del proyecto, deberemos definir y escoger el sistema de documentación que usaremos para el proyecto, teniendo en cuenta factores como los siguientes:

- Formatos de los documentos, según su tipología y métodos de acceso. Deberemos definir los formatos y plantillas elegidos para los diagramas de diseño, hojas de cálculo, hojas de seguimiento del proyecto, documentos que registren fallos o cambios en las especificaciones durante el desarrollo, documentos que definan la interfaz de usuario, etc.

- Método de acceso y flujo de trabajo de cada tipo de documento. Quién va a tener acceso a los diferentes tipos de documentos y bajo qué privilegios. Dónde se van a notificar los cambios que se realicen (¿en el propio documento?, ¿en un sistema de control de versiones?).

En el caso de la documentación del propio desarrollo (del código fuente del proyecto), conviene estudiar las herramientas que nos ofrezca el propio lenguaje para generar la documentación, ya que en muchos casos existen herramientas de generación de documentación a partir del código fuente y comentarios insertados mediante una sintaxis determinada que van a ayudar mucho en el proceso. En el capítulo 7 se examinan ejemplos de herramientas de este tipo.

ANOTACIONES

En el momento de tomar decisiones de formato y flujos de trabajo sobre la documentación, es de vital importancia tener en cuenta los estándares de formatos de documento existentes, evitando los formatos propietarios sobre todo en organizaciones heterogéneas donde convivan distintos sistemas operativos o en proyectos de software libre, para dar a la documentación la mayor accesibilidad posible. Suele ser una buena decisión escoger un formato de documentación fácilmente convertible a otros (p.ej., XML) y así poder disponer de la documentación en HTML para su consulta rápida, en PDF para agregar a la documentación del proyecto, etc.

1.4.7. Pruebas y calidad

La planificación de prototipos, pruebas y tests para el aseguramiento de la calidad es también un tema muy tratado en la ingeniería del software.

Para que un proyecto software tenga éxito, es necesario que el resultado cuente con la calidad esperada por el cliente o los usuarios. Así pues, la calidad del proyecto deberá poderse definir en términos de prestaciones, respuestas esperadas a determinadas acciones, o accesibilidad del producto en diferentes condiciones para poder probarla posteriormente mediante unos tests de calidad específicos.

Deberá ser posible realizar un plan de pruebas o de aseguramiento de la calidad que clasifique las actividades relacionadas con la calidad del producto según su importancia, y que defina con qué frecuencia y qué resultados se deberían obtener de cada una para pasar a la siguiente o para cumplir los requisitos para esa *release* en particular.

El proceso de aseguramiento de la calidad no trata únicamente de que el producto pase todos los tests establecidos, sino que implicará en muchos casos aspectos como:

- El uso de hojas de estilo aprobadas por los usuarios.
- La confección y repaso de *checklists* sobre funcionalidades.
- La revisión periódica del producto con los usuarios.

ANOTACIONES

- Las herramientas que pondremos a disposición de los usuarios para comunicar los errores o sugerencias.
- El uso de herramientas de análisis para medir la respuesta del proyecto a determinadas situaciones, o para simular un uso normal del mismo.

Los modelos tradicionales de ciclo de vida del software incluían la fase de pruebas como un proceso que había que llevar a cabo una vez finalizado el desarrollo. Esto se ha probado que resulta altamente contraproducente, no sólo por el coste de arreglar fallos o deficiencias una vez terminado el desarrollo, sino por la naturaleza evolutiva de muchos proyectos (sobre todo en entornos de software libre), en los que la fase de desarrollo no termina nunca estrictamente hablando.

Así pues, se tiende a incorporar las pruebas (o los mecanismos para llevarlas a cabo de modo automatizado) en el desarrollo desde el primer momento. Tecnologías como las pruebas unitarias o modelos como el pair-programming o el peer-testing nos ayudarán a mantener una disciplina en el desarrollo y a aumentar la calidad del resultado del proyecto.

Los tipos de pruebas, herramientas de gestión de las mismas y los planes de calidad se tratan en detalle en el capítulo 3.

1.4.8. Seguridad

La seguridad en proyectos software ha sido un factor clave desde el inicio de la ingeniería del software. Igual que en otros conceptos como la calidad, la seguridad no puede ser un módulo en el diseño que tiene lugar fuera de él, ni un proceso que se comprueba al finalizar el desarrollo, sino que se trata de un aspecto del proyecto que tiene que tenerse en cuenta y planificarse desde la fase de diseño, y que afectará a lo largo de todo el ciclo de vida del proyecto.

Los principios básicos de la seguridad de un sistema son los siguientes:

- Confidencialidad: los recursos (o funcionalidades sobre ellos) son accesibles sólo para los usuarios (o procesos) autorizados.

ANOTACIONES

- Integridad: los recursos pueden ser modificables sólo por los usuarios autorizados.

- Disponibilidad: los recursos accesibles están disponibles.

Muy frecuentemente, la seguridad de un proyecto irá más allá del mismo, afectando al sistema en el que éste se va a implantar, y por lo tanto deberemos asegurar que su implantación deja al sistema final en un estado seguro.

Es por esta razón por lo que, en la fase de diseño del proyecto, se debe tener en cuenta la seguridad del mismo, primero de forma general mediante un análisis de riesgos y de actividades destinadas a mitigarlos, y más adelante mediante la ampliación de los casos de uso según los principios básicos de la seguridad comentados anteriormente.

El análisis de riesgos consistirá, de forma muy resumida, en las siguientes actividades:

- Recopilación de los recursos que deben ser protegidos. La información, todo un sistema, un sólo dispositivo, etc.

- Clasificación de los actores del proyecto. Cuáles son sus casos de uso y qué roles representan.

- Recopilación de requisitos legales y de negocio. Certificaciones que hay que cumplir, restricciones de encriptación por exportación a determinados países o bien reglas de negocio más específicas como la aceptación o no de la repudiación por parte de los usuarios.

- Con la información recopilada, deberíamos ser capaces de construir una tabla en la que, para cada riesgo, estimáramos el coste que tiene por incidente, y a partir de una estimación de incidentes por año, nos permitiera decidir sobre la estrategia que hay que implantar (aceptar el riesgo tal como es y definir un plan de contingencia en caso de producirse, o bien mitigarlo

ANOTACIONES

mediante desarrollos adicionales, otras herramientas o cambios en el diseño que lo permitan).

Figura 6

ID riesgo	Recurso	Valor recurso	Coste por incidente	Incidentes/ año	Coste/ año	Estrategia
	Información del cliente	6.000 €				
1. Pérdida de información			6.000 €	2	12.000 €	Aceptar
2. Robo de información			60.000 €	1	60.000 €	Mitigar

Sin entrar en detalle, enunciamos a continuación un conjunto de buenas prácticas genéricas que ayudarán a mitigar los riesgos asociados a cualquier proyecto:

- Asignar el mínimo privilegio posible a cada actor en el sistema.
- Simplicidad. La seguridad por ofuscación no da buenos resultados.
- Diseño abierto. Siempre tendremos alternativas para mejorar o asegurar más el sistema.
- Seguridad por defecto. El sistema debe ser el máximo de seguro por defecto, si tiene que ser posible relajar las restricciones, debe tratarse de una acción adicional a realizar con el proyecto ya implantado.
- Fallada segura. Si el sistema falla o puede fallar, evitar que lo haga quedándose en una modalidad insegura.
- Minimizar el uso de recursos compartidos.
- Trabajar a favor de la usabilidad del proyecto redunda en un mejor uso de él y en menores probabilidades de fallos de seguridad por un mal uso del mismo.

ANOTACIONES

ANOTACIONES

1.5. Ejemplos de metodologías

Hasta ahora hemos hecho un breve repaso a la historia y los conceptos relacionados con la ingeniería del software en general. A continuación, vamos a examinar un poco más detalladamente dos metodologías concretas para ver cómo intentan resolver algunos de los problemas comentados, y poder, así, tomar decisiones en la gestión de proyectos.

Cabe comentar que no es siempre conveniente escoger y aplicar una metodología de forma estricta. Es importante entenderla y conocer qué nos puede aportar a nuestro proyecto, para aplicarla en esas fases o procesos en los que nuestro equipo o nuestros usuarios estén más cómodos con ella, y no al revés.

En entornos de software libre, no es habitual que los programadores que colaboran de forma altruista en el proyecto quieran hacerlo bajo una metodología demasiado estricta. Sin embargo, valorarán enormemente las facilidades de gestión que ofrezca el proyecto, el acceso a su documentación, la calidad de la misma, etc., por lo que estarán mucho más motivados, y pasarán más tiempo colaborando en el proyecto que entendiéndolo. De la misma manera, determinados colaboradores o clientes rechazarán una metodología demasiado moderna o flexible por desconocimiento.

1.5.1. Metodología próxima al software libre: eXtreme Programming

eXtreme Programming o programación eXtrema, es una de las metodologías llamadas “ágiles”, para el desarrollo de proyectos de software. Se basa en los principios de la simplicidad, la comunicación, la retroalimentación y el coraje para implicar a todo el equipo (y a los usuarios o clientes) en la gestión del proyecto. En 1996, Kent Back y Ward Cunningham pusieron en práctica una nueva metodología primando la simplicidad y evitando los hábitos que convertían las cosas fáciles en difíciles durante el desarrollo de un proyecto en DaimlerChrysler. El resultado fue la metodología eXtreme Programming o XP (que por supuesto nada tiene que ver con software de la compañía Microsoft).

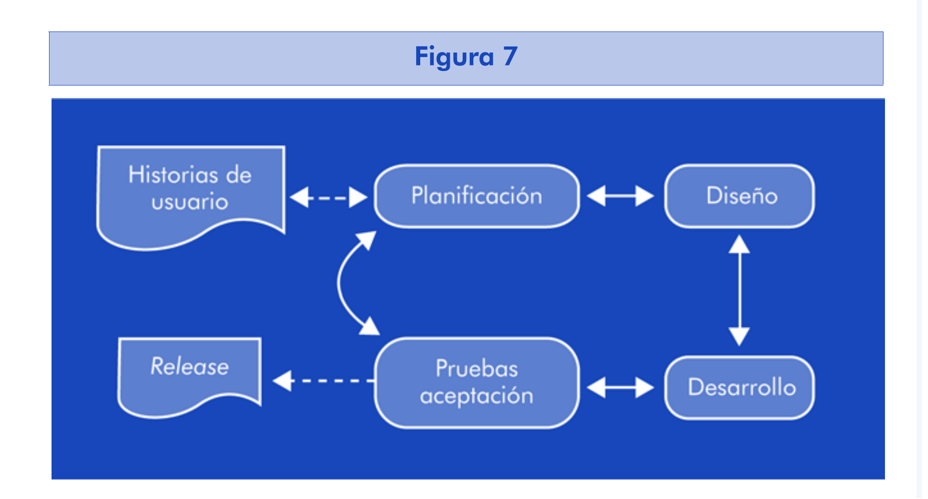
En su forma más genérica, las metodologías ágiles proponen una implicación total del cliente en el proyecto, y llevan al límite el modelo de desarrollo evolutivo en espiral. Esto es, realizar un plan de proyecto basado en versiones del producto acordadas a partir de funcionalidades concretas, y realizar el desarrollo para esas funcionalidades concretas. Una vez entregada la versión del proyecto cumpliendo con los requisitos (no un prototipo, sino una versión funcionando), el proceso vuelve a iniciarse con un conjunto mayor de funcionalidades.

Los procesos y prácticas de esta metodología están basados en la experiencia de equipos de desarrollo, y en los errores cometidos o encontrados una y otra vez al utilizar metodologías más tradicionales. Sin embargo, veremos que algunas de sus propuestas son bastante chocantes y otras serán incompatibles con grandes organizaciones o grandes proyectos.

La proximidad de esta metodología al entorno del software libre viene dada por sus principios básicos, más que por su aplicación concreta. En los próximos apartados examinaremos estos principios y veremos que algunos están muy cercanos al software libre, mientras que otros (sobre todo los que tienen que ver con el cliente, o con la programación en pares) no suelen ser demasiado aplicables.

eXtreme Programming, puede dividirse en cuatro principios sobre los que se va iterando hasta que el proyecto ha finalizado (el cliente aprueba el proyecto). Estas fases o principios son **planificación**, **diseño**, **desarrollo** y **pruebas**. Aunque a primera vista parece que no estamos añadiendo nada nuevo a las metodologías tradicionales, es en los detalles de cada fase, y en los objetivos que nos marcaremos en cada una de ellas (iteración tras iteración) donde están las mayores diferencias.

Figura 7



ANOTACIONES

Planificación

La planificación empieza con la confección de las “Historias de usuario”. De manera similar a los casos de uso, se trata de breves frases escritas por el cliente (no más de tres líneas), en las que se describe una prestación o un proceso sin ningún tipo de tecnicismo (es el usuario o el cliente quien las escribe).

Estas historias de usuario servirán para realizar la planificación de *releases*, así como para los tests de aceptación con el cliente. Para cada historia deberíamos poder estimar su tiempo ideal de desarrollo, que debería ser de 1, 2 o 3 semanas como máximo. Si el tiempo de desarrollo es mayor, deberemos partir la historia en trozos que no excedan de esas estimaciones.

A continuación podemos pasar a la propia planificación de la próxima (o primera) *release* del proyecto. En la reunión de planificación deberemos implicar al cliente, al gestor del proyecto y a los desarrolladores. El objetivo será planificar las siguientes *releases* poniendo en orden las historias de usuario que faltan por desarrollar. Deberá ser el cliente quien dicte el orden de las historias de usuario, y los desarrolladores quienes estimen el tiempo que les llevaría idealmente en desarrollarlo (idealmente aquí significa sin tener en cuenta dependencias, ni otros trabajos o proyectos, pero sí incluyendo las pruebas).

Las historias de usuario se suelen reflejar en tarjetas o trozos de papel, que se van agrupando y clasificando encima de la mesa durante la planificación. El resultado deberá ser un conjunto de historias que tengan sentido, y que puedan llevarse a cabo en poco tiempo. Para planificarlo, podemos hacerlo según dos criterios, basándonos en el tiempo hasta la siguiente *release* o en el alcance (entendido como el número de funcionalidades) que deseamos que tenga.

Aquí introducimos un nuevo concepto, la *velocidad* del proyecto. Esta velocidad nos servirá para decidir cuántas historias de usuario vamos a incluir en la próxima *release* (si estamos planificando en base al tiempo, ya fijado), o bien cuando vamos a tardar en lanzar la *release* (si estamos planificando por alcance, ya fijado). La velocidad del proyecto es simplemente el número de historias de usuario completadas en la iteración anterior. Si se trata de la primera iteración, habrá que hacer una estimación inicial.

ANOTACIONES

La velocidad puede aumentarse, si en el transcurso del desarrollo se finalizan las historias de usuario previstas antes de tiempo. Entonces los desarrolladores pedirán historias de usuario que estaban planeadas para la siguiente *release*. Este mecanismo permite a los desarrolladores recuperarse en los periodos duros, para después acelerar el desarrollo si es posible. Recordemos que las historias deben tener todas una duración máxima, y que cuanto más parecido sea el volumen de trabajo estimado en cada una de ellas, más fiable será la medida de velocidad del desarrollo.

Este método de planificación rápido y adaptativo nos permite hacer un par de iteraciones para tener una medida fiable de lo que será la velocidad media del proyecto y estimar así más detalladamente el plan de *releases* (además de haber empezado ya con el desarrollo del mismo) en el intervalo de tiempo en que otras metodologías tardarían en documentar, planificar y realizar una estimación completa, que quizá no sería tan fiable.

En la reunión de planificación, al inicio de cada iteración, también habrá que incorporar las tareas que hayan generado los tests de aceptación que el cliente no haya aprobado. Estas tareas se sumarán también a la velocidad del proyecto, y el cliente, que es quien escoge las historias de usuario que hay que desarrollar, no podrá elegir un número mayor que el de la velocidad del proyecto.

Los desarrolladores convertirán las historias de usuario en tareas (esas sí en lenguaje técnico) de una duración máxima de tres días ideales de programación cada una. Se escribirán en tarjetas y se pondrán encima de la mesa para agruparlas, eliminar las repetidas, y asignarlas a cada programador. Es de vital importancia evitar añadir más funcionalidades que las que la historia de usuario estrictamente requiera. Esta tendencia de los gestores de proyectos o analistas acostumbrados a las metodologías tradicionales debe evitarse en modelos iterativos como éste, ya que desvirtúan las estimaciones y el principio de *releases* frecuentes.

Con las tareas resultantes, se volverá a comprobar que no estamos excediendo la velocidad del proyecto, eliminando o añadiendo historias de usuario hasta llegar a su valor. Es posible que historias de usuario diferentes tengan tareas en común, y esta fase de la planificación nos permitirá filtrarlas, para así poder aumentar la velocidad del proyecto añadiendo más historias de usuario en esta iteración.

ANOTACIONES

En el momento de planificar el equipo de trabajo encargado de cada tarea o historia, es importante la movilidad de las personas, intentar que cada dos o tres iteraciones todo el mundo haya trabajado en todas las partes del sistema. Conceptos como el *pair programming*, que veremos en siguientes apartados, también pueden ayudar.

Diseño

Durante el diseño de la solución, y de cada historia de usuario que lo requiera, la máxima simplicidad posible es la clave para el éxito de esta metodología. Sabiendo que un diseño complejo siempre tarda más en desarrollarse que uno simple, y que siempre es más fácil añadir complejidad a un diseño simple que quitarla de uno complejo, siempre deberemos hacer las cosas lo más sencillas posible, evitando añadir funcionalidad no contemplada en esa iteración.

Asimismo, es importante y se ha probado muy útil encontrar una metáfora para definir el sistema. Es decir, un proceso o sistema que todos conozcan (el cliente, el gestor del proyecto, los desarrolladores) y que puedan identificar con el proyecto que se está desarrollando. Encontrar una buena metáfora ayudará a tener:

- Visión común: todo el mundo estará de acuerdo en reconocer dónde está el núcleo del problema, y en cómo funciona la solución. Nos ayudará a ver cómo será el sistema o qué podría llegar a ser.

- Vocabulario compartido: la metáfora nos ayudará a sugerir un vocabulario común para los objetos y procesos del sistema. Dependiendo de la metáfora escogida, el vocabulario puede ser altamente técnico o por el contrario muy común.

- Generalización: la metáfora puede sugerir nuevas ideas o soluciones. Por ejemplo, en la metáfora "el servicio de atención al cliente es una cadena de montaje", nos sugiere que el problema va pasando de grupo en grupo de gente. Pero también nos hace pensar en qué pasa cuando el problema llega al final de la cadena...

- Arquitectura: la metáfora dará forma al sistema, nos identificará los objetos clave y nos sugerirá características de sus interfaces.

ANOTACIONES

Para el diseño del sistema, se sugiere llevar a cabo reuniones entre todos los desarrolladores implicados, donde se toman las decisiones mediante el uso de tarjetas CRC (*class*, *responsabilities* y *collaboration*). Cada tarjeta representa un objeto del sistema, en la que su nombre aparece escrito en la parte superior, sus responsabilidades en la parte izquierda y los objetos con los que colabora en la parte derecha.

Figura 8

Nombre de clase:	Superclase:	Subclases:
Responsabilidad principal:		
Responsabilidades:	Colaboraciones:	

El proceso de diseño se realiza creando las tarjetas (al inicio sólo escribiremos el nombre en ellas, el resto ya se irá completando) y situándolas próximas a las que comparten interfaces o llamadas. Las tarjetas correspondientes a objetos que heredan o son interfaces de otros pueden situarse encima o debajo.

Este mecanismo ayuda a que todo el mundo participe y aporte sus ideas en el diseño moviendo las tarjetas encima de la mesa según éste va progresando. Si se llega a un punto muerto y se decide volver a empezar, será tan simple como "limpiar" la mesa y volver a ir situando las tarjetas. No tendremos que borrar diagramas ni hacer trabajo extra diseñando alternativas. Cuando tengamos el diseño definitivo, también será más fácil de mantener en la memoria por parte de los participantes y es entonces cuando entraremos en detalle en cada objeto y haremos los diagramas necesarios.

Finalmente, es clave también el concepto de refactorización, es decir, el hecho de realizar cambios necesarios en las partes de código que lo requieran, sin modificar su funcionalidad o su interacción con el resto del sistema. A medida que vayamos avanzando en iteraciones en el proyecto, nos veremos obligados a modificar o ampliar partes de código ya escritas anteriormente. En ese momento, en lugar de dejar lo que funcionaba sin tocarlo y desarrollar el módulo adicional para la nueva funcionalidad, deberemos hacer el esfuerzo de

ANOTACIONES

refactorizar el módulo existente, dejándolo igual de simple pero con la nueva funcionalidad añadida. Será siempre mucho más fácil de probar, de explicar y de comprender para el resto del equipo.

Este concepto, provocará cambios en el diseño que habíamos hecho en iteraciones anteriores, pero esto no es perjudicial, sino al contrario. Deberemos darnos cuenta de que el diseño evoluciona iteración tras iteración, por lo que el diseño anterior ya estará obsoleto a partir de ese momento, y acostumbrarnos a ello.

Codificación

Una primera diferencia importante entre esta metodología y las llamadas “tradicionales” es la disponibilidad del cliente, que debe ser total. En lugar de limitarse a escribir durante semanas una hoja de requisitos, el cliente debe participar en las reuniones de planificación, tomar decisiones, y estar disponible para los desarrolladores durante los tests funcionales y de aceptación.

Debido a la movilidad de los desarrolladores durante el proyecto, participando en cada iteración en partes distintas del mismo, es de vital importancia acordar unos estándares de codificación y respetarlos en el desarrollo. Cada lenguaje de programación tiene sugerencias o reglas más o menos detalladas al respecto. Deberemos ser tan concretos como sea posible, no dejando al libre albedrío temas como la indentación en el código, la sintaxis y nombres de variables, etc. En casi todos los casos existen unas convenciones recomendadas por los creadores del lenguaje que cubren estos aspectos.

Siempre es conveniente tener en cuenta estas recomendaciones y utilizarlas tal cual o bien adaptarlas a las preferencias de nuestros programadores, pero en cualquier caso es imprescindible definir las decisiones tomadas al respecto.

Por lo que respecta a la propia codificación en sí, eXtreme Programming nos propone que antepongamos la creación de los tests unitarios al propio desarrollo de las funcionalidades. Los tests unitarios se explicarán con detalle en el capítulo correspondiente, pero básicamente son pequeños trozos de código que prueban las funcionalidades de un objeto, de modo que esa prueba pueda incorporarse a un

ANOTACIONES

proceso de pruebas automatizado. La mayoría de los lenguajes tienen hoy en día librerías para crear y ejecutar pruebas unitarias.

La idea que se fragua detrás de esta propuesta es que creando primero las pruebas que deberá pasar nuestro código, tendremos una idea más clara de lo que deberemos codificar, y nos guiaremos por ello, implementando únicamente lo que nos permita pasar la prueba. También tendremos ventajas adicionales al conocer cuándo hemos acabado de implementar la funcionalidad requerida, o en la comprensión de la misma.

Otra característica diferencial de eXtreme Programming es el pair programming o la programación por parejas. Se ha demostrado que dos programadores, trabajando conjuntamente, lo hacen al mismo ritmo que cada uno por su lado, pero el resultado obtenido es de mucha más calidad. Simplemente los dos sentados frente al mismo monitor, y pasándose el teclado cada cierto tiempo, provoca que mientras uno está concentrado en el método que está codificando, el otro piense en cómo ese método va a afectar al resto de objetos y la interacción, las dudas y propuestas que surgen reducen considerablemente el número de errores y los problemas de integración posteriores.

En una metodología incremental como ésta, la integración con lo que ya se ha desarrollado en iteraciones anteriores es clave. No hay una única solución a ese problema. Dependiendo del proyecto y del equipo de trabajo, se pueden proponer soluciones, como por ejemplo designar “dueños” para cada objeto, que conozcan todos los tests unitarios y de integración que debe pasar, y se ocupe de integrarlo en cada *release*. Otra alternativa es que cada pareja de programadores se hace responsable de integrar cuando todos los tests de la tarea que estaban realizando pasan al 100%. De este modo, añadimos al paradigma *release-often* (distribuye o implanta a menudo) el *integrate-often* (integra a menudo), reduciendo enormemente las posibilidades de problemas de integración.

Pruebas

Ya hemos hablado sobre las pruebas unitarias en la fase anterior, y se verán en detalle en el apartado correspondiente. Es importante

ANOTACIONES

insistir en este tema, ya que, aun en el caso de tener una fecha de entrega muy próxima, la construcción de las pruebas unitarias va a ahorrarnos mucho más tiempo del que invertimos programándolas, buscando pequeños fallos y protegiéndonos contra ellos de forma permanente durante el resto del desarrollo.

Cuanto más complicada sea la prueba, más necesaria es para asegurar que después el desarrollo hace lo que se requiere.

A veces existe la tendencia a pensar que las pruebas unitarias pueden hacerse durante los últimos tres meses de desarrollo. Esto es erróneo, ya que sin esas pruebas, el desarrollo se alarga esos tres meses, y quizá más.

Los tests unitarios ayudarán también a la refactorización, ya que asegurarán que los cambios que hayamos introducido en la iteración actual no afecten a la funcionalidad.

Cuando encontremos un fallo o *bug* en los tests de aceptación con el cliente, o durante el uso, deberemos crear un test unitario que lo compruebe. Así aseguramos que no vuelve a surgir en siguientes iteraciones.

Los tests de aceptación se crearán a partir de las historias de usuario. Una historia puede tener uno o varios tests, según sea la funcionalidad que hay que probar. El cliente es responsable de definir los tests de aceptación, que deberían ser automatizables en la medida de lo posible. Estos test son del tipo "caja negra", en el sentido de que únicamente nos definen el resultado que debe tener el sistema ante unas entradas concretas. Los tests de aceptación que no tengan éxito generarán nuevas tareas para la próxima iteración, afectando así a la *velocidad* del proyecto, y proporcionando además una puntuación del éxito o fracaso de cada historia de usuario, o de cada equipo de trabajo.

Conclusiones

Hemos visto de manera rápida e introductoria la metodología eXtreme Programing, y comentado sus particularidades respecto a las metodologías tradicionales. Sin duda, introduce algunos conceptos novedosos

ANOTACIONES

y hasta discutibles como el pair-programming, pero también usa técnicas más convencionales como las tarjetas CRC.

Prescindiendo de los detalles, lo que XP propone es un cambio de paradigma a lo largo de toda la metodología. Las herramientas concretas, como las pruebas unitarias, las historias de usuario, la refactorización, etc. no son más que recursos que también podríamos utilizar en otras metodologías. Lo verdaderamente destacable de XP es su forma de ordenar el ciclo de vida del proyecto, y la involucración con el cliente. Sin esto, no estamos haciendo XP.

Sus principales características lo hacen muy adecuado para proyectos de software libre, aunque, como hemos visto en su breve historia, no fue concebido con este objetivo. Otros aspectos como el pair-programming o el diseño colaborativo con tarjetas CRC no son tan aplicables en este ámbito.

1.5.2. Metodología clásica: Métrica v3

"Métrica v3" es la metodología de planificación, desarrollo y mantenimiento de sistemas de información promovida por la Secretaría del Consejo Superior de Informática del Ministerio de Administraciones Públicas, que es el órgano interministerial responsable de la política en informática del Gobierno español.

Aunque su ámbito inicial es el de las administraciones públicas, las mejoras introducidas en la versión 3 y el mejor uso de estándares y normas de ingeniería del software hacen que su alcance pueda ampliarse a las administraciones autonómicas, locales y al resto de empresas y organizaciones.

Entre las mejoras introducidas en la versión 3.0 (publicada en el año 2000), destaca la incorporación de nuevos métodos y tecnologías (cliente/servidor, interfaz gráfica de usuario, orientación a objetos), así como la incorporación de aspectos de gestión (que la metodología denomina *interfaces*) para mejorar aspectos que no pertenecen a una sola fase, sino que intervienen a lo largo del proyecto, como son la gestión del mismo, la calidad y la seguridad, entre otros.

ANOTACIONES

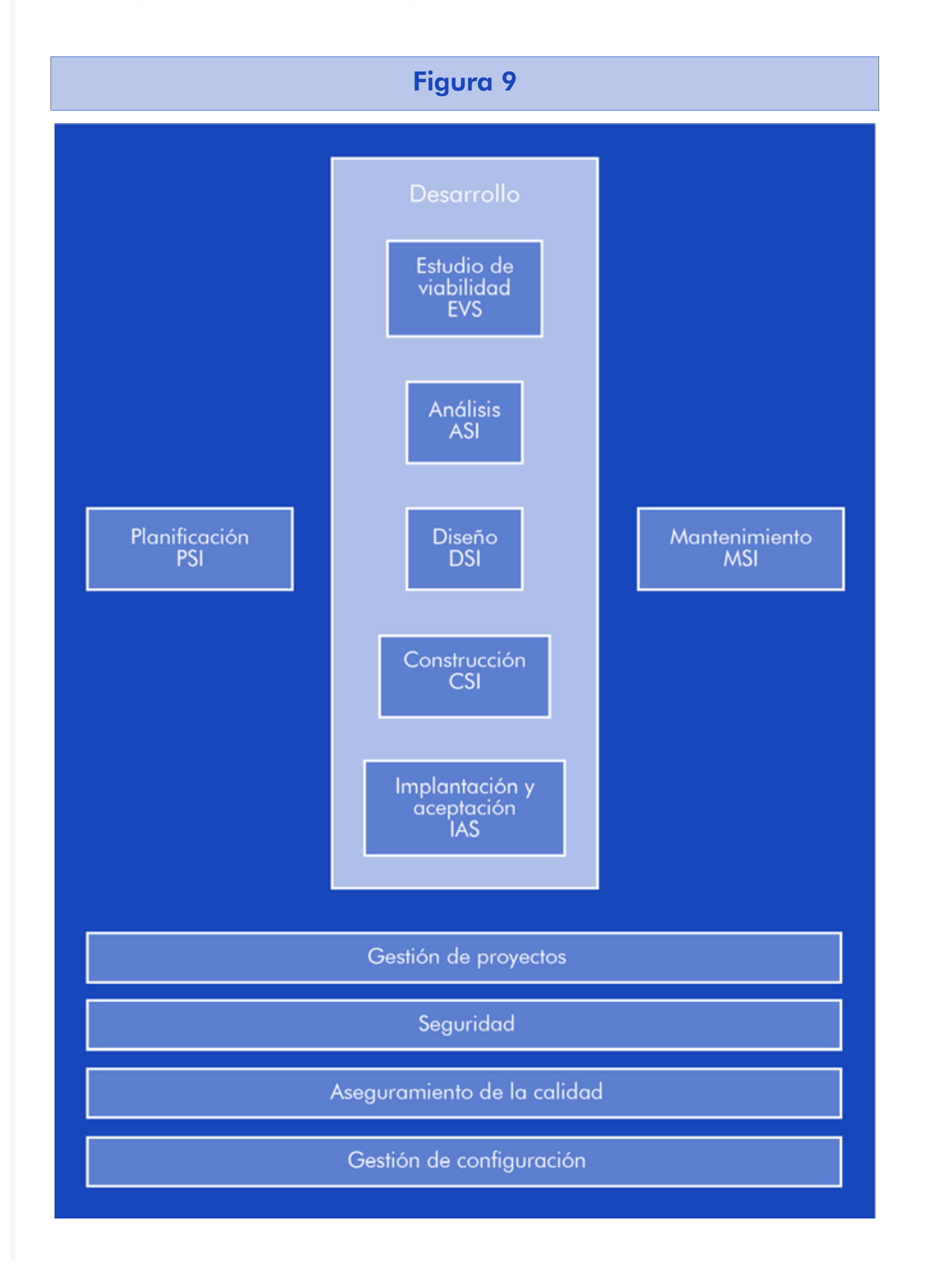
La estructura de la metodología sigue el clásico método en cascada basado en los siguientes procesos:

- Planificación
- Desarrollo
- Mantenimiento

Cada proceso de los anteriores detalla las actividades y tareas que hay que realizar, de manera que para cada tarea se indican:

- Las técnicas y prácticas a utilizar.
- Los responsables de realizarla.
- Sus productos de entrada y salida.

Figura 9



ANOTACIONES

El aspecto más destacable de esta metodología no es tanto lo que pueda aportar como innovación a la ingeniería del software en sí, sino el esfuerzo que se ha hecho por poner a disposición pública una metodología completa, más o menos actualizada, y que representa un marco inicial de referencia para presentar proyectos a la Administración pública (que lo exigen como requisito), pero que podemos adaptar a nuestra empresa o proyecto en el sector privado, si nuestra organización se siente más cómoda con modelos clásicos de desarrollo.

Métrica v3 define muy bien los documentos de entrada de cada proceso, actividad y tarea, así como el resultado que genera. A lo largo de éste y los siguientes apartados, vamos a destacar los más relevantes. Si se desea ampliar información, la documentación disponible es muy extensa, y existen ejemplos, cursos de autoformación, así como programas auxiliares de ayuda y selección de herramientas compatibles con la metodología.

Planificación de sistemas de información

Este proceso tiene como objetivo último la creación del Plan de sistemas de información (PSI) de la organización. Adaptando el marco y los objetivos, podemos utilizar sus actividades para generar el plan del proyecto en concreto en el que estamos trabajando.

Entre las actividades que deberemos realizar, destacan:

- Descripción de la situación actual.
- Conjunto de modelos que constituye la arquitectura de la información.
- Priorización y calendario de los proyectos a desarrollar.
- Evaluación de los recursos necesarios.
- Plan de seguimiento y cumplimiento, bajo una perspectiva estratégica y operativa.

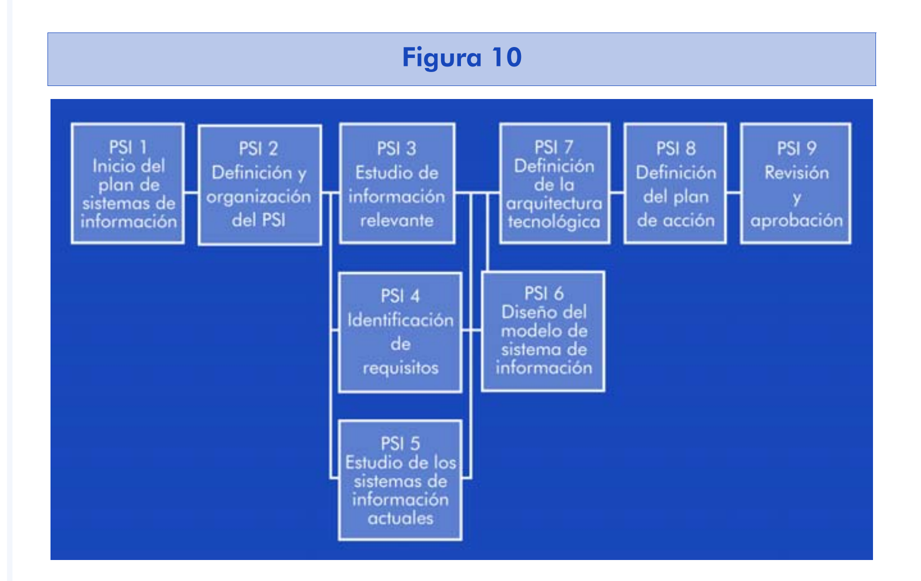
El plan debe ser realizado a alto nivel, sin tecnicismos y con una perspectiva estratégica y operativa. Asimismo, es fundamental que la dirección se implique en su desarrollo. Para la descripción de la

ANOTACIONES

situación actual, el nivel de detalle dependerá de la documentación de que se disponga y de la predisposición de la organización a una sustitución total o parcial del sistema de información actual.

El cuadro completo de actividades que hay que llevar a cabo durante esta fase es el siguiente:

Figura 10



Actividades

1. Inicio del Plan de sistemas de información

El objetivo de esta primera actividad es la obtención de la descripción general del Plan de sistemas de información, identificando los objetivos generales en los que se centra, y el ámbito al que afecta dentro de la organización.

Se deberá identificar también a los participantes en la elaboración del plan, que definirán los factores críticos de éxito del mismo.

El método de obtención de esta información es la participación en sesiones de trabajo del comité de dirección hasta que nombre a los gestores del proyecto.

2. Definición y organización del plan

Una vez se han determinado los responsables del proyecto, y sus objetivos, deberemos detallar el alcance del plan, organizar el equipo de personas que lo van a llevar

ANOTACIONES

a cabo y elaborar un calendario de ejecución. Este calendario deberá incluir una valoración en términos económicos a partir de estimaciones, que permitan tomar decisiones en cuanto a su aprobación.

Una vez definido el plan de trabajo para la realización del propio plan, se deberá comunicar a la dirección para su aprobación definitiva.

3. Estudio de la información relevante

La primera actividad, una vez aprobado el plan, deberá ser la recopilación y análisis de todos los antecedentes generales que puedan afectar a los procesos y recursos contemplados en el plan, así como los resultados que presentaron. De especial interés serán los estudios realizados con anterioridad, relativos a los sistemas de información del ámbito del plan como a su entorno tecnológico.

La información que obtengamos será de utilidad para realizar la toma de requisitos en actividades posteriores.

4. Identificación de requisitos

La toma de requisitos se realizará mediante el estudio de los procesos en el ámbito del plan. El modelo de estos procesos, junto con sus actividades y funciones, la información implicada en ellos y las unidades organizativas o recursos que en él participan, se obtendrá de reuniones de trabajo con usuarios expertos o técnicos implicados.

Una vez contrastadas las conclusiones, se elaborará el modelo de procesos implicados, unificando en la medida de lo posible los que guarden relación entre ellos, con el objetivo de tener una visión lo más general posible.

A continuación se deberán analizar las necesidades de información de cada proceso modelado anteriormente, y se elaborará un modelo de información que refleje las principales entidades y las relaciones existentes entre ellas en términos de información de entrada/salida, sus actividades y sus funciones.

ANOTACIONES

A continuación se deberán analizar las necesidades de información de cada proceso modelado anteriormente, y se elaborará un modelo de información que refleje las principales entidades y las relaciones existentes entre ellas en términos de información de entrada/salida, sus actividades y sus funciones.

Finalmente, elaboraremos el catálogo de requisitos a partir de la información obtenida en actividades anteriores, y de las necesidades de información y proceso obtenidos previamente. Es importante priorizar los requisitos, sobre la base de las opiniones de los usuarios y los objetivos del plan.

5. Estudio de los sistemas de información actuales

A partir de los sistemas actuales afectados por el plan, se deberá obtener una valoración de la situación actual, basada en criterios como la facilidad de mantenimiento, documentación, flexibilidad, facilidad de uso, nivel de servicio, etc. Los usuarios aportarán aquí los elementos de valoración más importantes.

Es importante obtener una valoración lo más objetiva posible, ya que ésta va a influir en la decisión de mejora o sustitución de cada proceso o sistema.

6. Diseño del modelo de sistemas de información

En este punto tendremos información suficiente para decidir en qué sistemas aplicamos mejoras o bien qué sistemas sustituimos, y en cada caso, cuál deberá ser el sistema resultante.

Una vez tomadas estas decisiones, debemos obtener el modelo de sistemas de información, que incluirá un diagrama de representación de todos ellos, con sus conexiones e interfaces, y una descripción de cada sistema con el conjunto de procesos y requisitos que cubre.

ANOTACIONES

7. Definición de la arquitectura tecnológica

En esta actividad debemos proponer una arquitectura tecnológica, a alto nivel, que dé soporte al modelo de sistemas de información, y que puede incluir, si es necesario, opciones. Para esta actividad tendremos en cuenta sobre todo los requisitos de carácter tecnológico, aunque puede ser necesario comprender el catálogo completo de requisitos.

La definición y elección entre las alternativas posibles deberá realizarse sobre la base del entorno actual, los estándares, y apoyándonos en análisis coste/beneficio e impacto en la organización de cada alternativa.

8. Definición del plan de acción

El plan de acción será el que definirá los proyectos concretos a llevar a cabo para la implantación de los sistemas y modelos de información definidos en las actividades anteriores.

Dentro del plan de acción, se incluirá un calendario de proyectos, con posibles alternativas y una estimación de recursos. Para la elaboración de este plan, será importante tener en cuenta las prioridades que habremos marcado en materia de requisitos, y los sistemas implicados en cada uno de ellos.

Por último, también será importante la realización de un plan de mantenimiento y control de la ejecución de los proyectos.

9. Revisión y aprobación del plan

Finalmente, debemos presentar la arquitectura de información diseñada y el plan de acción a los responsables de la dirección del mismo. Mejoraremos la propuesta si es necesaria y obtendremos la aprobación final.

ANOTACIONES

Desarrollo de sistemas de información

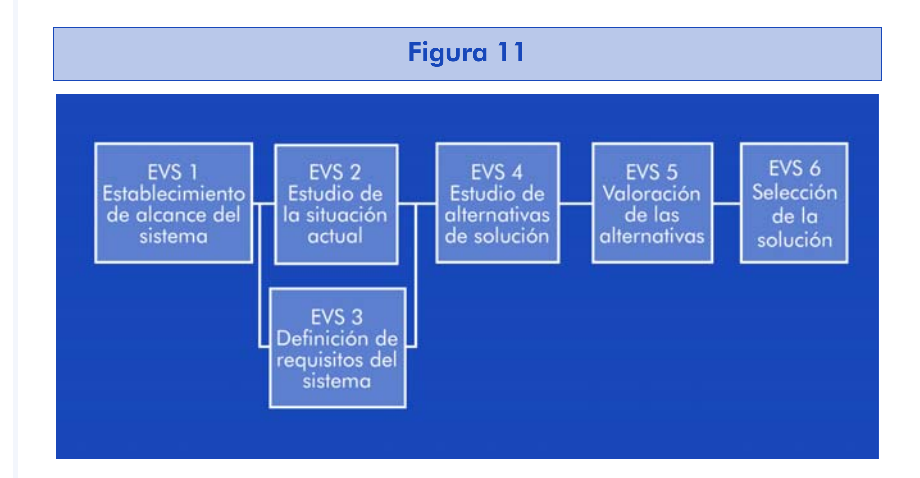
El plan de sistemas tenía como objetivo proporcionar un marco estratégico que sirva de referencia. Una vez completado el plan de acción, pasaremos al desarrollo de cada proyecto.

En él, algunas actividades serán réplicas de actividades realizadas en el plan de sistemas (toma de requisitos, análisis de la situación actual, etc.). Si el plan se ha realizado con el suficiente nivel de detalle, o se refería únicamente a un proyecto concreto, estas actividades no serán necesarias. En caso contrario, serán las primeras en llevarse a cabo en este proceso.

Estudio de viabilidad

Si el plan de sistemas nos ha dejado con varias alternativas para un proyecto en concreto, en la primera fase deberemos estudiar la viabilidad de cada una, en términos de impacto en la organización e inversión a realizar.

Figura 11



En todo caso, la primera actividad que Métrica v3 define en la fase de desarrollo es el estudio de la viabilidad del proyecto, que debería generar uno o varios documentos (según las alternativas consideradas) con un índice como el que sigue:

Solución propuesta:

Descripción de la solución
Modelo de descomposición en subsistemas

ANOTACIONES

Matriz procesos / Localización geográfica
Matriz datos / Localización geográfica
Entorno tecnológico y comunicaciones
Estrategia de implantación global del sistema
Descripción de procesos manuales

Si la alternativa incluye desarrollo:

Modelo abstracto de datos / Modelo de procesos
Modelo de negocio / Modelo de dominio

Si la alternativa incluye un producto software estándar de mercado:

Descripción del producto
Evolución del producto
Costes ocasionados por el producto
Estándares del producto
Descripción de adaptación (si es necesaria)
Contexto del sistema (con la definición de las interfaces)
Impacto en la organización de la solución
Coste / Beneficio de la solución
Valoración de riesgos de la solución
Enfoque del plan de trabajo de la solución
Planificación de la solución

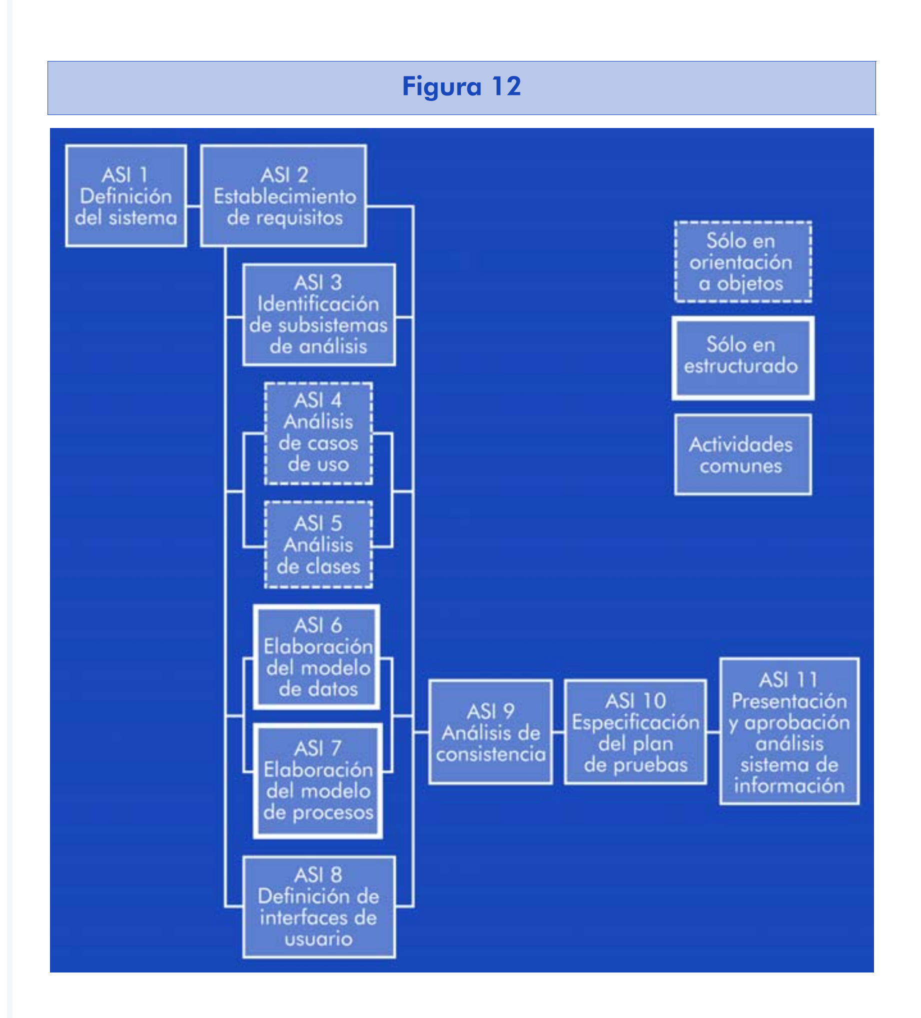
Análisis del sistema de información

El objetivo de este proceso es la obtención de una especificación detallada del sistema de información que satisfaga las necesidades de los usuarios y sirva de base para el posterior diseño del sistema.

Métrica v3 soporta el desarrollo tanto con lenguajes estructurados, como orientados a objetos, pero las actividades particulares en cada caso están integradas en una estructura común.

ANOTACIONES

Figura 12



Las dos primeras actividades serán recuperar y profundizar en la definición del sistema y la toma de requisitos del mismo, con el objetivo de detallar al máximo el catálogo de requisitos y describir con precisión el sistema de información.

A continuación empezará el cuerpo del análisis, formado por cuatro actividades que se realimentarán entre sí, hasta tener el análisis completo del sistema. De las cuatro actividades, dos son comunes para desarrollos estructurados u orientados a objetos, mientras que las otras dos son diferentes en cada caso.

Las actividades son:

- Identificación de subsistemas de análisis: facilitar el análisis mediante la descomposición del sistema en subsistemas. Actividades posteriores pueden obligar a revisar esta descomposición.

- Análisis de casos de uso (elaboración del modelo de datos en lenguaje estructurado).
- Análisis de clases (elaboración del modelo de procesos en lenguaje estructurado).
- Definición de interfaces de usuario.

Finalmente, deberemos verificar y validar los modelos con el fin de asegurar que son:

- Completos: cada modelo contiene toda la información necesaria.
- Consistentes: cada modelo es coherente con el resto.
- Correcto: el modelo sigue las normas de calidad, estándares y nomenclatura determinadas en actividades anteriores.

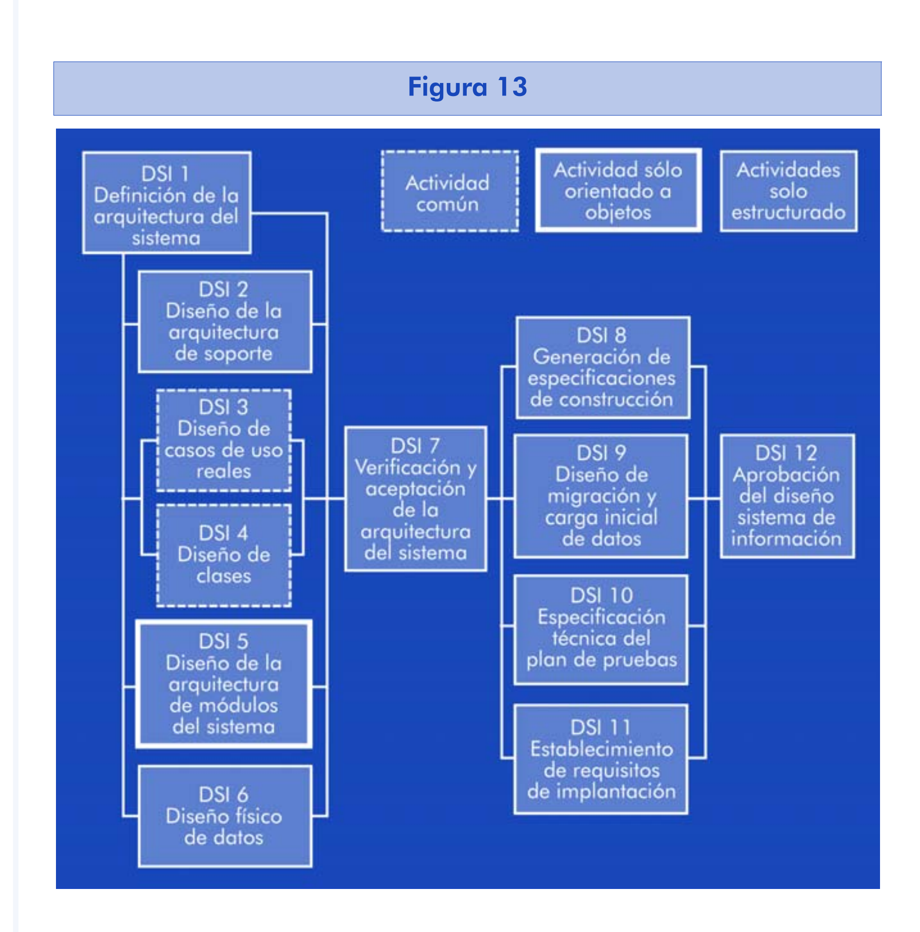
Como última actividad del análisis, se deberá especificar el plan de pruebas del sistema. Se trata únicamente de iniciar su especificación, que se detallará en actividades posteriores. Contemplando el alcance de las mismas, los requisitos del entorno de pruebas, y la definición de las pruebas de aceptación es suficiente.

Diseño del sistema de información

El objetivo de esta actividad es la especificación detallada de la arquitectura del sistema y del entorno tecnológico que le va a dar soporte, junto con los componentes del sistema de información.

ANOTACIONES

Figura 13



A partir de esta información, generaremos las especificaciones de construcción del sistema, así como la descripción técnica del plan de pruebas, la definición de requisitos de implantación y el diseño de procedimientos de migración y carga inicial si procede.

Las actividades de este proceso se dividen en dos bloques:

El primer bloque de actividades se realiza en paralelo, y comprende:

- Definición de la arquitectura del sistema de información: se establece la partición física del sistema de información, y su correspondencia con los subsistemas de diseño. También definiremos sus requisitos de operación, administración, seguridad y control de acceso. Los subsistemas de diseño se deberán clasificar en:

– Subsistemas de soporte: contienen elementos o servicios comunes al sistema y a la instalación. Generalmente originados por la interacción con la infraestructura técnica, o por la reutilización.

ANOTACIONES

– Subsistemas específicos: contienen los elementos propios del sistema de información, como continuidad de los vistos en la actividad de análisis del sistema.

• Diseño de casos de uso reales: es el diseño detallado del comportamiento del sistema para los casos de uso, junto con la interfaz de usuario.

• Diseño de clases: detallado con atributos, operaciones, relaciones, métodos y la estructura jerárquica de todo el modelo.

• Diseño físico de datos una vez obtenido el modelo de clases.

En el segundo bloque de actividades, se generan todas las especificaciones necesarias para la construcción:

• Generación de especificaciones de construcción que fijan las directrices para la construcción de los componentes del sistema.

• Diseño de la migración y carga inicial de datos.

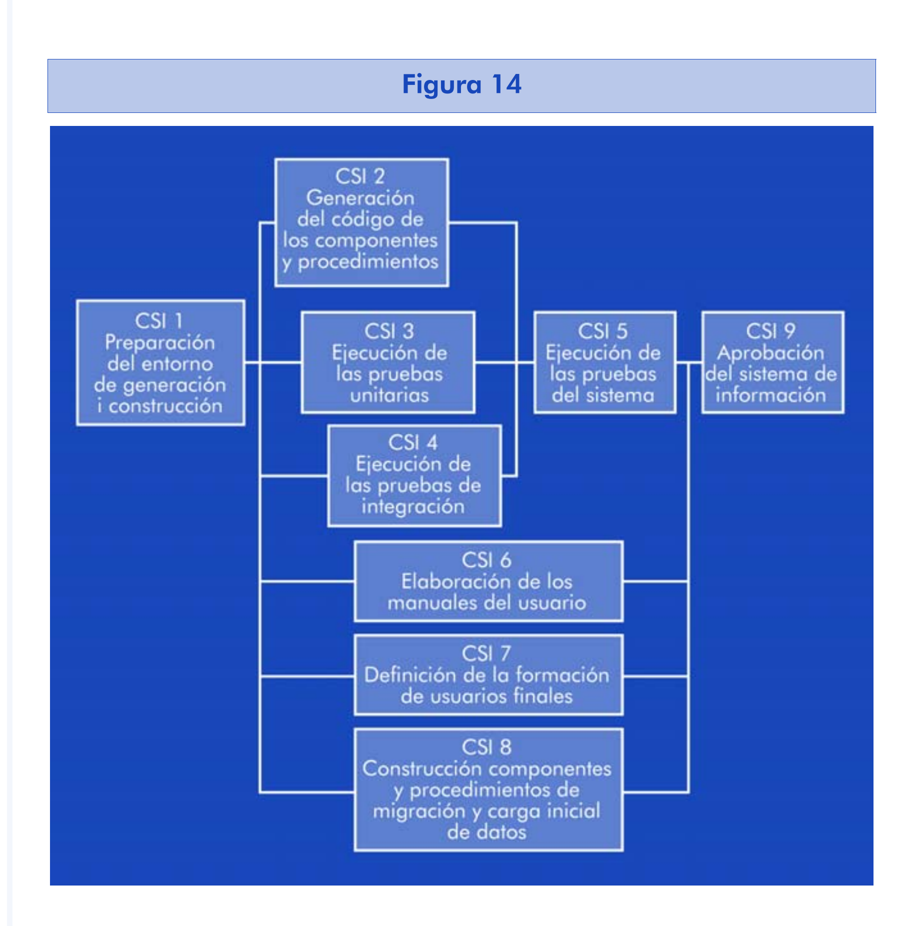
• Especificación técnica del plan de pruebas. Realizar un catálogo de excepciones permitirá establecer un conjunto de verificaciones relacionadas con el diseño o con la propia arquitectura.

• Establecimiento de requisitos de implantación.

Construcción del sistema de información

En este proceso se genera el código de los componentes del sistema de información, se desarrollan todos los procedimientos de operación y seguridad y se elaboran los manuales de usuario y de explotación.

Figura 14



Un objetivo clave en esta fase será asegurar el correcto funcionamiento del sistema para su aceptación y posterior implantación. Para conseguirlo, en este proceso se realizarán las pruebas unitarias, las pruebas de integración de los subsistemas y las pruebas de sistema, de acuerdo con el plan establecido.

La base para la construcción del sistema es la especificación de construcción obtenida en el proceso de diseño anterior. Una vez configurado el entorno de desarrollo, se realizará la codificación y las pruebas de acuerdo con las siguientes actividades:

- Generación del código: conforme a las especificaciones de construcción y al plan de integración de los subsistemas.

- Ejecución de las pruebas unitarias: conforme al plan de pruebas diseñado.

- Ejecución de las pruebas de integración: verificaciones asociadas a los componentes y subsistemas.

ANOTACIONES

- Ejecución de pruebas de sistema: integración final del sistema de información, comprobando tanto las interfaces entre subsistemas y sistemas externos, como los requisitos.

- Elaboración del manual de usuario: documentación de usuario final y de explotación.

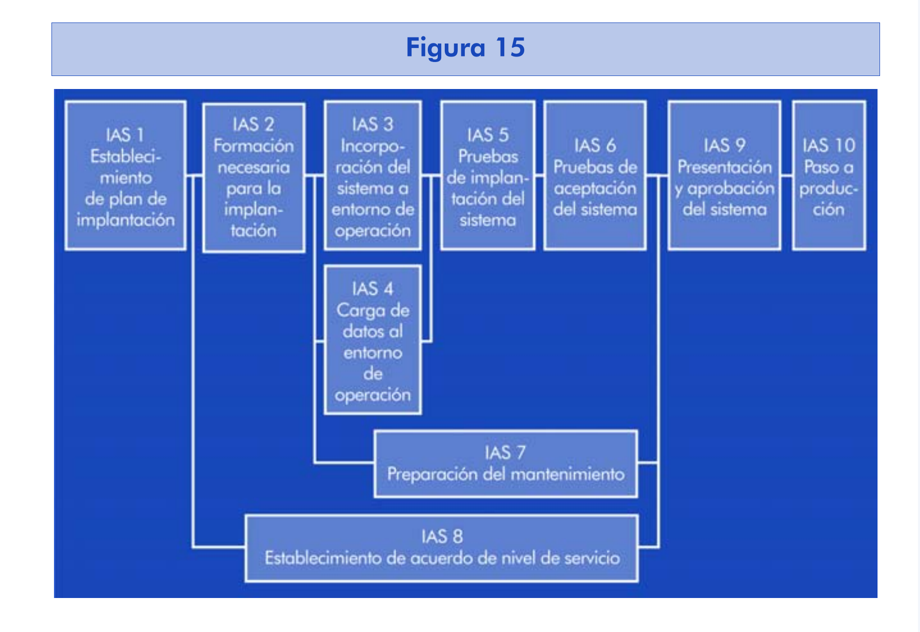
- Formación de usuarios finales.

- Construcción de los componentes y procedimientos de migración y carga inicial de datos.

Implantación y aceptación del sistema

El objeto de este proceso es la entrega y aceptación del sistema en su totalidad, así como la realización de las actividades necesarias para su paso a producción.

Figura 15



En éste, deberemos:

- Revisar la estrategia de implantación determinada en el estudio de viabilidad del sistema.

- Preparar la infraestructura necesaria, la instalación de los componentes, la activación de los procedimientos manuales y automáticos, y la migración o carga inicial de datos.

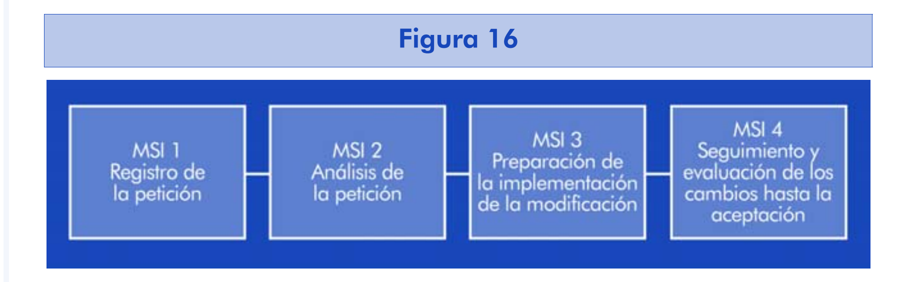
ANOTACIONES

- Realizar las pruebas de implantación y aceptación del sistema en su totalidad.
- Preparar el mantenimiento.
- Determinar los requisitos de los servicios que requiere el sistema. Habrá que distinguir entre servicios de operaciones (seguridad, comunicaciones, etc.) y servicios al cliente (atención al usuario, mantenimiento, etc.).
- El plan de implantación puede definir este proceso de forma iterativa, poniendo en marcha los subsistemas y realizando estas actividades para cada uno de ellos.

Mantenimiento de sistemas de información

El proceso de mantenimiento tiene como objeto la puesta en marcha de una nueva versión del sistema o proyecto a partir de las peticiones de los usuarios con motivo de un problema detectado en el sistema o por una necesidad de mejora del mismo.

Figura 16



Una vez recibida una petición, ésta se incorpora a un catálogo de peticiones, y se procede a diagnosticar de qué tipo de mantenimiento se trata:

- Correctivo: cambios precisos para corregir errores.
- Evolutivo: modificaciones necesarias para cubrir la expansión o cambio en las necesidades.
- Adaptativo: modificaciones que afectan al entorno en que el proyecto opera, cambios de configuración, de hardware, bases de datos, comunicaciones, etc.

ANOTACIONES

• Perfectivo: acciones llevadas a cabo para mejorar la calidad interna de los sistemas. Reestructuración de código, optimización de rendimiento, eficiencia, etc.

El siguiente paso que define Métrica v3 es la determinación de la responsabilidad en atender la petición. La petición puede ser aceptada o rechazada, en cuyo caso se registra y termina el proceso. En caso de aceptarse, habrá que estudiar la viabilidad del cambio, verificar y reproducir el problema y estudiar el alcance de la modificación.

El plazo y urgencia de la petición debe establecerse según los parámetros del estudio anterior. Mediante este análisis, la persona encargada del mantenimiento deberá valorar el coste y esfuerzo necesario para la implementación de la modificación.

Las tareas de procesos de desarrollo se corresponden con las de los procesos de **análisis**, **diseño**, **construcción** e **implantación**. Finalmente, es importante establecer un plan de pruebas de regresión que asegure la integridad del sistema afectado.

El registro de las peticiones recibidas puede utilizarse también para fines estadísticos (peticiones atendidas, sistemas afectados, etc.), y un registro minucioso de las actividades realizadas, una documentación extensa de los cambios incorporados al sistema repercutirá directamente en la calidad de los sistemas resultantes y en tener el coste del mantenimiento bajo control.

Interfaces

Al ser Métrica v3 una metodología fuertemente estructurada, y a pesar de que su aplicación proporciona sistemas con calidad y seguridad, se han definido unas interfaces que refuerzan éstos y otros aspectos a lo largo de todos sus procesos. Las interfaces descritas en la metodología son:

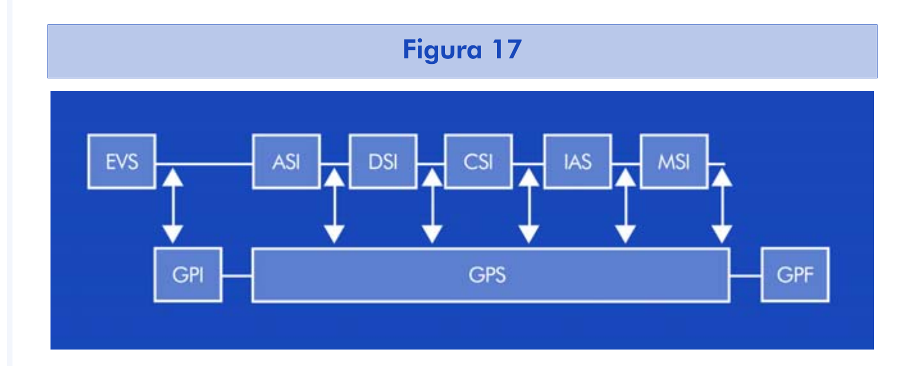
• Gestión de proyectos
• Seguridad
• Aseguramiento de la calidad
• Gestión de la configuración

ANOTACIONES

Gestión de proyectos

La gestión de proyectos tiene como finalidad principal la planificación, el seguimiento y control de las actividades y de los recursos humanos y materiales que intervienen en el desarrollo del sistema de información o del proyecto. El objetivo de este control es la identificación en todo momento de los problemas que se produzcan y de su resolución o la capacidad de mitigarlos de forma inmediata.

Figura 17



- **GPI**: Las actividades de inicio del proyecto se realizarán al concluir el estudio de viabilidad del mismo, y consistirán en la estimación del esfuerzo y en la propia planificación del proyecto. Para la estimación del esfuerzo, partiremos de la descomposición en subsistemas obtenido del estudio de viabilidad y de los elementos implicados (funciones, entidades y datos en desarrollos estructurados o clases, propiedades y métodos en desarrollos orientados a objeto).

  Para el cálculo del esfuerzo, Métrica v3 propone la utilización de técnicas conocidas como el método Albrecht o los puntos de función (en desarrollo estructurado) o bien la métrica Staffing Size en el caso de desarrollo orientado a objetos.

  Métrica v3 no da recomendaciones en cuanto a la estrategia de desarrollo, simplemente enumera las existentes (en cascada, por subsistemas, por prototipo o híbrida) y deja la elección en manos del jefe de proyecto.

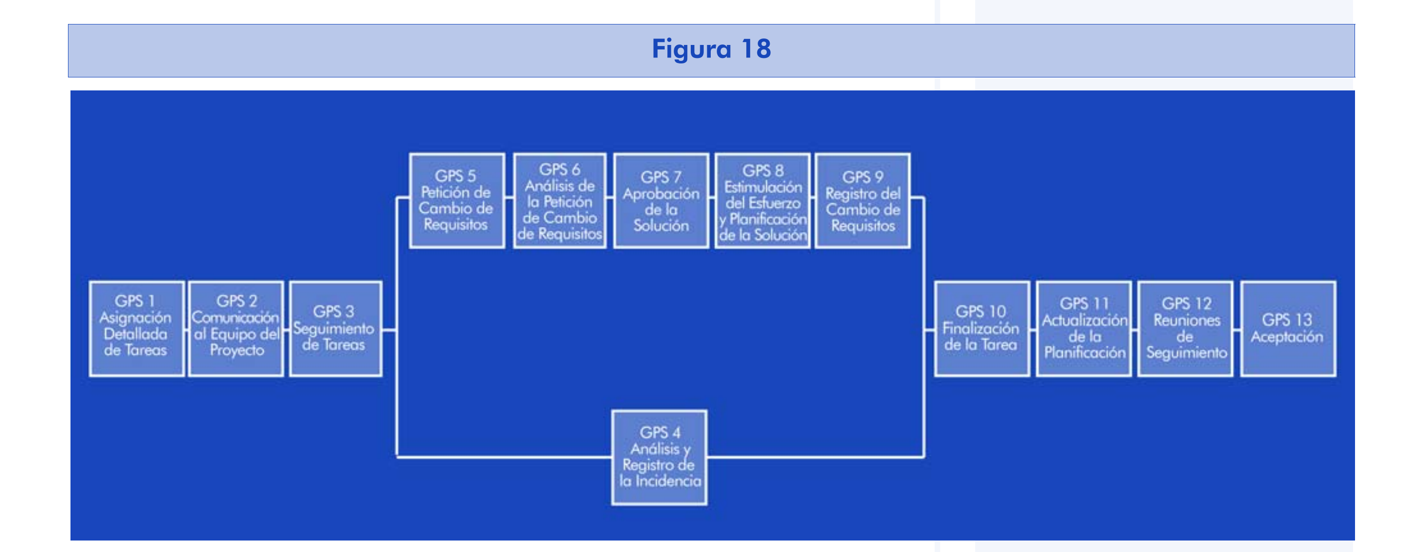
  En cuanto a la descomposición detallada de actividades y la asignación de recursos, Métrica v3 recomienda el uso del método PERT y los diagramas de Gantt, conjuntamente con tablas de asignación de recursos.

ANOTACIONES

• **GPS**: Las actividades de seguimiento y control comprenden desde la asignación de tareas hasta su aceptación interna, incluyendo la gestión de incidencias, cambios en los requisitos y la vigilancia del correcto desarrollo de las tareas y actividades establecidas en la planificación.

  Para ello, Métrica v3 enumera las actividades que deben tener lugar en paralelo al análisis, diseño, construcción, implantación y mantenimiento del proyecto. Las resumimos en la siguiente figura.

Figura 18



• **GPF**: Por último, al concluir el proyecto, se realizan las actividades propias de cierre del mismo y registro de la documentación que incluye.

  El cierre del proyecto consiste en resumir todos los datos de éste que se consideren de interés, para que puedan servir de referencia a proyectos futuros, aprovechando las experiencias habidas y tratando de incurrir en los mismos errores.

Seguridad

El objetivo de esta interfaz es incorporar mecanismos de seguridad adicionales a los que se proponen como parte de la propia metodología, estableciendo un conjunto de actividades que tienen lugar a lo largo de todo el proceso.

ANOTACIONES

Dentro de la interfaz, existen dos tipos de actividades diferenciadas:

- Actividades relacionadas con la seguridad intrínseca del sistema que se va a desarrollar. Son actividades de análisis en detalle de los requisitos de seguridad que tienen lugar durante la planificación, el estudio de viabilidad y el análisis y diseño del sistema.

- Actividades que cuidan de la seguridad del propio proceso de desarrollo.

Si en la organización ya existe un plan de seguridad o metodología de análisis y gestión de riesgos, deberá aplicarse para detectar aquellas necesidades que no se encuentren cubiertas y tomar las decisiones en función de los riesgos que se quiera asumir o mitigar.

Si no existe un plan de seguridad, habrá que desarrollarlo desde el principio. El plan deberá recoger las medidas de seguridad activas, preventivas y reactivas en respuesta a situaciones en las que pueda producirse un fallo, reduciendo su efecto, tanto si estamos tratando con el sistema de información o proyecto en sí como durante el proceso de desarrollo.

Aseguramiento de la calidad

El objetivo de esta interfaz es proveer un marco de referencia para la definición y puesta en marcha de planes específicos de aseguramiento de la calidad del proyecto. Lo que se pretende es dar confianza en que el producto cumple con los requisitos.

Métrica v3 recomienda que el grupo de aseguramiento de la calidad sea totalmente independiente del de desarrollo, y tendrá como misión identificar las posibles desviaciones en los estándares, requisitos y procedimientos establecidos, así como comprobar que se han llevado a cabo las medidas preventivas o correctoras necesarias.

Para un resultado óptimo de esta interfaz, deberá aplicarse desde el estudio de la viabilidad del sistema y a lo largo de todo el desarrollo, en los procesos de análisis, diseño, construcción, implantación y mantenimiento del proyecto.

ANOTACIONES

Gestión de la configuración

El objetivo de esta interfaz es crear y mantener un registro de todos los productos creados o usados durante el proyecto. Sus actividades tienen lugar a lo largo de todo el proyecto, se encargarán de mantener la integridad de los productos, de asegurarnos de que cada persona involucrada dispone de la última versión de todos ellos, y de que disponemos de toda la información acerca de cambios producidos en sus configuraciones, requisitos, etc.

Al hablar de productos, no nos referimos únicamente a código fuente y ejecutables, también a todo el material y documentación generados durante el proyecto, diagramas, manuales, estudios y análisis que se hayan revisado, etc.

Nos ayudará a valorar el impacto de cada cambio en el sistema, y a reducir su tiempo de implementación. El cumplimiento y coste de implementación de esta interfaz no deben menospreciarse, si la organización ya disponía de un sistema de gestión de configuraciones, sus actividades se desarrollarán con más facilidad y rapidez.

La información que podría solicitarse al sistema de gestión de configuración es muy amplia:

- Información sobre una fase del proyecto concreta, análisis, construcción, etc.
- Información sobre la "vida" de un producto, sus versiones, personas involucradas, etc.

Sus actividades se distribuirán a lo largo de las distintas fases, una vez determinada la selección de la solución en el estudio de viabilidad, se seleccionará el entorno tecnológico donde se registrará la configuración del proyecto, y de ahí en adelante se pondrá en marcha el proceso.

En cada una de las fases siguientes, análisis, diseño, etc. se irán registrando los cambios en los productos, y al final de cada fase se anotarán en el registro global.

ANOTACIONES

Evidentemente, en el proceso de mantenimiento, esta interfaz adquiere una importancia especial, ya que la información de que dispondremos al valorar cada nuevo cambio reducirá y compensará lo invertido a lo largo de las otras fases. El trabajo también será intenso, ya que nuevas versiones se sucederán más rápidamente y habrá que incorporarlas y notificarlas al registro de configuraciones.

Conclusiones

Tanto por su lenguaje, como por el tipo y número de actividades planteadas, se detecta rápidamente que Métrica v3 es una metodología impulsada por una administración pública. Tal como comentamos al inicio, su mérito radica en la propia elaboración de la metodología y no tanto en sus planteamientos, que son de lo más clásico.

Si nos sentimos cómodos con este tipo de metodologías, un aspecto muy interesante es la documentación que proporciona sobre las técnicas (documentos, plantillas, listas de comprobación, etc.) que cabe aplicar en cada actividad. También es muy conveniente la definición concreta que realiza sobre las entradas y salidas de cada actividad. Aunque nuestra metodología basada en Métrica v3 no siga al pie de la letra todas las actividades planteadas, un estudio de los documentos que deberemos obtener y utilizar en cada fase puede ser de gran ayuda para asegurarnos de que no nos dejamos nada antes de pasar a la siguiente fase del proyecto.

1.6. Resumen

A lo largo de este capítulo, hemos dado un repaso general a la ingeniería del software. Hemos visto los problemas que tuvieron los primeros gestores de grandes proyectos, y conocido los conceptos que deben tenerse en cuenta en el momento de plantearse un proyecto software de cualquier envergadura.

Los dos ejemplos de metodologías planteados, aunque muy diferentes, intentan solucionar los mismos problemas, y organizaciones o proyectos diferentes se encontrarán más próximos a una u otra indistintamente.

ANOTACIONES

A lo largo del resto de capítulos, veremos la aplicación práctica de muchos de los conceptos introducidos aquí en el ámbito concreto del software libre y los estándares.

1.7. Otras fuentes de referencia e información

Albrecht, A.J.; Gaffney S.H. (1983). *Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation*

Beck, K. (1999). *Extreme Programming Explained*. Addison-Wesley Pub Co.

Brooks, F.P. (1995). *The Mythical Man-Month*.

Consejo Superior de Informática. *Métrica v3*. Ministerio de Administraciones Públicas. http://www.csi.map.es/csi/metrica3/

CSE. Center for Software Engineering COCOMO II. http://sunset.usc.edu/research/COCOMOII/

Development Support Center. http://www.functionpoints.com/

Djikstra, E. *The Humble Programmer*. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF

Extreme Programming: A gentle introduction. http://www.extremeprogramming.org/

Libre Software Engineering. http://libresoft.dat.escet.urjc.es

Lorenz, M.; Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice Hall Inc.

Naur, P.; Randell, B. (1968). *Software Engineering*. Report on conference sponsored by the Nato Science Comité.

Pair Programming. Wikipedia. http://en.wikipedia.org/wiki/Pair_programming

ANOTACIONES

PERT.
http://en.wikipedia.org/wiki/Program_Evaluation_and_Review_Technique

Pressman, R.S. (2004). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science.

Raymond, E.S. (1997). "The Cathedral and the Bazaar". http://www.catb.org/~esr/writings/cathedral-bazaar/

ReadySET. http://readyset.tigris.org/

The Future of Software Engineering. http://www.softwaresystems.org/future.html

Wikipedia. The Free Encyclopedia. http://www.wikipedia.org

ANOTACIONES

2. Diseño de software orientado a objeto con UML

2.1. Introducción

El análisis y diseño del software constituye una parte fundamental en cualquier proyecto, independientemente de su tamaño. Todas las metodologías, en mayor o menor medida, o con distintos alcances, dan una gran importancia a estas fases como paso intermedio entre la toma de requisitos y el desarrollo del proyecto.

Como parte de lo que se considera "Ingeniería del software", el modelado y diseño ha ido evolucionando a lo largo del tiempo mediante técnicas aportadas tanto por especialistas del ámbito académico, como por empresas especializadas en consultoría y desarrollo.

Existen muchas técnicas orientadas a modelar un aspecto de los sistemas de información, bases de datos, interfaces de usuario, componentes, flujos de datos, etc., pero pocas han aportado un enfoque global al problema.

A finales de los años noventa, una empresa en particular (Rational Corp.) empezó una iniciativa para desarrollar un estándar de modelado a la que se sumaron científicos y otras empresas del sector. Así nació UML (*Unified Modeling Language*), que hoy en día sigue siendo el método de modelado más completo y aceptado en la industria.

El método de modelado está basado en el paradigma de programación orientado a objetos, que en ese momento empezaba a popularizarse y hoy en día sigue siendo (con alguna variación y revisión) el más usado en todo tipo de proyectos.

Por ello, en este capítulo empezaremos comentando este paradigma de programación y sus aspectos fundamentales. No se ha pretendido hacer un tratado completo sobre la orientación a objetos, sino proporcionar una base para comprender los conceptos que soportará cada tipo de diagrama.

ANOTACIONES

A continuación presentaremos un ejemplo práctico que iremos siguiendo a lo largo del capítulo y aplicando en cada apartado correspondiente un tipo de diagrama UML.

Finalmente, comentaremos otra utilidad de UML, la generación de código a partir de ciertos tipos de diagramas. Presentaremos tres herramientas de código abierto muy populares y veremos sus prestaciones en este ámbito.

2.2. Objetivos

Los objetivos que el estudiante deberá haber logrado al finalizar el capítulo de "Diseño de software orientado a objeto con UML" del material "Ingeniería del software en entornos de software libre" son los siguientes:

- Tener claros los conceptos más importantes relacionados con la orientación a objetos tanto a nivel de su análisis y diseño como a nivel de implementación.
- Conocer los distintos tipos de diagramas UML, sus cometidos y particularidades.
- Disponer de los conocimientos necesarios para afrontar el análisis y diseño de un proyecto software y representar su modelo mediante UML.
- Conocer algunas herramientas de modelado y generación de código e identificar la más idónea para un proyecto concreto.

2.3. Revisión de conceptos del diseño orientado a objeto y UML

El presente apartado tiene como objetivo introducir los conceptos del análisis, el diseño y la programación orientada a objeto necesarios para poder asimilar el resto del capítulo y entender el caso práctico que plantearemos.

ANOTACIONES

2.3.1. Introducción a la orientación a objetos

La orientación a objetos es un paradigma más de programación en el que un sistema se expresa como un conjunto de objetos que interactúan entre ellos. Un paradigma de programación nos proporciona una abstracción del sistema real a algo que podemos programar y ejecutar, y puede decirse que el tipo de abstracción está directamente relacionada con los problemas que puede resolver o al menos con la facilidad con que podremos resolverlos. Mientras que el lenguaje ensamblador es una abstracción del procesador, podríamos decir que otros lenguajes de programación como BASIC o C son abstracciones del propio lenguaje ensamblador. Aunque han supuesto un importante avance sobre éste, aún obligan a los programadores a pensar en términos de la estructura del ordenador en lugar de la estructura del problema que están intentando solucionar.

Otros lenguajes han intentado modelar el problema presentando visiones diferentes del mundo (todos los problemas son listas en LISP, todos los problemas son una cadena de decisiones en PROLOG). Estas visiones son buenas soluciones para el tipo de problemas que se ajustan a esa visión, pero cuando nos salimos de ella, el paradigma empieza a no ser tan útil.

La orientación a objetos da un paso más y nos proporciona las herramientas para representar elementos en el espacio del problema concreto. No nos impone ninguna restricción ***a priori***, de forma que el programador no está limitado a cierta clase de problemas. Los elementos en el espacio del problema y su representación es lo que llamamos "objetos".

Alan Kay resumió las cinco características básicas de Smalltalk, el primer lenguaje de programación puramente orientado a objetos, en el que están basados tanto C++ como Java:

- **Todo es un objeto**: pensemos en un objeto como una "gran variable"; almacena datos, pero también podemos hacerle peticiones para que haga operaciones sobre ella misma. En teoría, podemos escoger cualquier componente conceptual del problema que queremos resolver (estudiantes, edificios, servicios, etc.) y representarlo como un objeto en nuestro programa.

ANOTACIONES

- **Un programa es un conjunto de objetos que se dicen unos a otros lo que deben hacer mediante mensajes**: para hacer una petición a un objeto, le mandamos un mensaje. Podemos pensar en el mensaje como la petición para llamar a un método que pertenezca a un objeto (enviarNota, activarAlarma, imprimeInforme).

- **Cada objeto tiene su propia memoria hecha de otros objetos**: dicho de otra forma, crearemos un nuevo tipo de objeto empaquetando otros objetos existentes. Así podremos construir programas de mucha complejidad escondiéndola en simples objetos.

- **Cada objeto tiene un tipo**: o dicho de otra forma, cada objeto es una instancia de una clase. Aquí utilizamos *clase* como sinónimo de "tipo". La característica más importante de una clase es "¿Qué mensajes podremos enviarle?".

- **Todos los objetos de un tipo particular pueden recibir los mismos mensajes**: al ser un objeto de tipo "círculo", también un objeto de tipo "figura", un círculo podrá recibir mensajes para figuras. Esto significa que podemos escribir código que hable con "figuras" y automáticamente estaremos preparados para hablar con cualquier objeto de ese tipo.

La definición del paradigma no ha llegado a consensuarse nunca, pero en su forma más general podríamos decir que expresa el hecho de ver el programa en términos de las cosas (los objetos) que manipula, en lugar de las acciones que realiza. Hoy en día, entendemos por un lenguaje orientado a objeto el que nos proporciona las siguientes prestaciones:

- **Objetos**: empaquetan los datos y la funcionalidad conjuntamente. Son la base para la estructura y la modularidad en un software orientado a objetos.

- **Abstracción**: la habilidad de un programa para ignorar ciertos aspectos de la información que está manipulando. Cada objeto del sistema es un modelo de un "actor" que puede trabajar en el sistema, informar o cambiar su estado y comunicarse con otros objetos, sin revelar cómo están implementadas estas prestaciones.

ANOTACIONES

- **Encapsulación**: asegurar que los usuarios de un objeto no pueden cambiar su información o estado de formas permitidas. Sólo los propios métodos que ofrece el objeto deben poder cambiar su información. Cada objeto ofrece una interfaz que especifica cómo el resto de objetos deben trabajar con él. El objetivo de esta encapsulación es mantener la integridad del objeto.

- **Polimorfismo**: diferentes objetos pueden tener la misma interfaz para responder al mismo tipo de mensaje, y hacerlo apropiadamente según su naturaleza.

- **Herencia**: organiza y facilita el polimorfismo permitiendo a los objetos definirse como especializaciones de otros, que pueden compartir y extender su funcionalidad sin tener que reimplementarlo de nuevo. Esto suele hacerse agrupando los objetos en clases, y definiendo otras clases como extensiones de éstas, creando árboles de clases.

También puede decirse que la definición de la orientación a objetos proviene del "objeto gramatical" de una acción. Veamos algunos ejemplos de cómo una funcionalidad se expresa de forma diferente en programación estructurada (orientada al sujeto) o en programación orientada a objetos:

- **Orientada al sujeto**: la aplicación de ventas guarda la transacción.

- **Orientada al objeto**: la transacción se salva a partir de un mensaje de la aplicación de ventas.

- **Orientada al sujeto**: la aplicación de ventas imprime la factura.

- **Orientada al objeto**: la factura se imprime a partir de un mensaje de la aplicación de ventas.

Como vemos, estamos asociando la acción al objeto al que afecta, no a quien la inicia o la pide.

ANOTACIONES

2.3.2. Historia

El concepto de "objeto" en programación apareció en 1967 cuando se presentó el lenguaje Simula'67 diseñado para programar simulaciones. La idea de agrupar datos y funcionalidad en una única estructura se originó en respuesta a la complejidad de representar todas las combinaciones de diferentes atributos que los elementos participantes en una simulación podían presentar. Más adelante, llegó SmallTalk como un refinamiento del mismo concepto desarrollado en el XeroxPARC que permitía la creación y uso de objetos en tiempo de ejecución.

Pero la popularización del paradigma tuvo lugar durante la década de los ochenta con la llegada de C++, una extensión del lenguaje C. Y su consolidación llegó con el éxito de las interfaces de usuario gráficas, a las que este paradigma se ajusta perfectamente.

Desde entonces, características de la orientación a objetos se han añadido a lenguajes de programación ya existentes como Ada, BASIC, Lisp, Perl, etc., lo que comportó problemas de incompatibilidad y dificultades en el mantenimiento de código. Por el contrario, a los lenguajes orientados a objeto "puros" les faltaban prestaciones a las que los programadores se habían acostumbrado.

En las últimas décadas, el lenguaje Java ha tenido un gran éxito no sólo por su implementación muy sólida del paradigma, también por su parecido a C y C++ y por el uso de una máquina virtual que permitía la ejecución de código en múltiples plataformas. La iniciativa .NET de Microsoft es parecida en este aspecto soportando también varios lenguajes de programación tanto de nueva creación como variantes de lenguajes ya existentes.

Más recientemente, han aparecido lenguajes que son principalmente orientados a objetos pero compatibles con metodologías estructurales como PHP, Python y Ruby.

De la misma forma que la programación procedural evolucionó hacia la programación estructurada, la orientación a objetos ha ido incorporando mejoras como los patrones de diseño y los lenguajes de modelado como UML.

ANOTACIONES

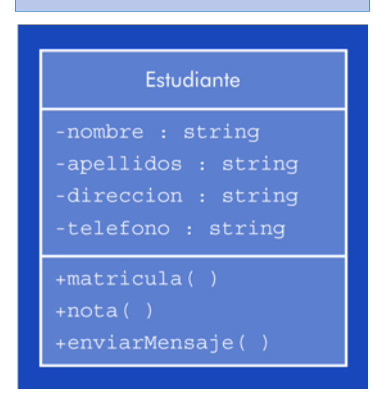
2.3.3. Clases y objetos

Como se ha comentado en apartados anteriores, todo objeto es instancia de una clase, entiendo ésta como la abstracción de un "tipo" de datos dentro del espacio del problema que queremos resolver.

Ya que una clase describe un conjunto de objetos que tienen características idénticas (elementos de datos o atributos) y comportamientos idénticos (funcionalidades o métodos), una clase es un tipo de datos a todos los efectos. Cada objeto de la clase dentro de nuestro programa tendrá unos valores concretos para sus datos (de ahí que llamemos "instancia" al objeto de la clase). Una vez definida la clase que representa un elemento de nuestro problema, podremos crear tantos objetos de ella como queramos y manipularlos para que se comporten como deben hacerlo en el problema que estamos resolviendo.

Cada objeto va a satisfacer o aceptar un cierto tipo de peticiones, según sea su cometido o responsabilidad en el problema. Las peticiones que podemos hacer a un objeto se definen en su "interfaz". Los atributos propios del objeto junto con el código que satisface esas peticiones es su "implementación".

Figura 1. Representación gráfica de una clase



El nombre de la clase es "Estudiante", los atributos del mismo son su "nombre", "apellidos", "dirección" y "teléfono" y las peticiones que podemos hacerle son "matrícula", "nota" y "enviarMensaje".

Nota

Esta figura sigue la notación definida en el *Lenguaje Unificado de Modelado* (UML), en el que la clase se representa mediante una caja dividida en tres zonas. El nombre se representa en la parte superior, los datos o atributos que deseemos mostrar en su parte central y las peticiones o métodos públicos en su parte inferior.

ANOTACIONES

En el momento de desarrollar o entender un problema que vamos a solucionar con orientación a objetos, suele ser útil pensar en los objetos en términos de "proveedores de servicios". Nuestro programa va a prestar servicios a los usuarios, y lo va a hacer utilizando servicios que presten los objetos. El objetivo es crear (o reutilizar) los objetos que proporcionen los servicios más ajustados a las necesidades del problema.

Siguiendo con el ejemplo del estudiante, si queremos que nuestro programa imprima su expediente imaginaremos un objeto "estudiante" similar al anterior, un objeto que calculará su expediente y un objeto que creará el informe con el formato adecuado para imprimirlo (quizá mediante otro objeto).

Al pensar de esta forma, inconscientemente estamos creando un sistema bien cohesionado. Esto es fundamental en la calidad del diseño del software, significa que los objetos (y sus interfaces) encajan bien unos con otros. En un buen diseño orientado a objeto, cada uno hace una cosa bien, y no intenta hacer demasiado. En el ejemplo, probablemente descubriríamos que el objeto que nos proveerá el servicio de impresión tendremos que dividirlo para poder proveer servicios de impresión genéricos, comunicarse con modelos específicos de impresoras, etc. También este ejercicio es útil para identificar objetos susceptibles de ser reutilizados o de que los encontremos ya desarrollados en catálogos públicos de objetos.

2.3.4. Encapsulación

Al diseñar los objetos de nuestra aplicación, necesitamos tener en cuenta los tipos de consumidores de los mismos. Por una parte tendremos a los programadores que van a poder crear clases (creadores de clases), y por la otra a los que sólo van a usarlas en la misma u otras aplicaciones (consumidores de clases).

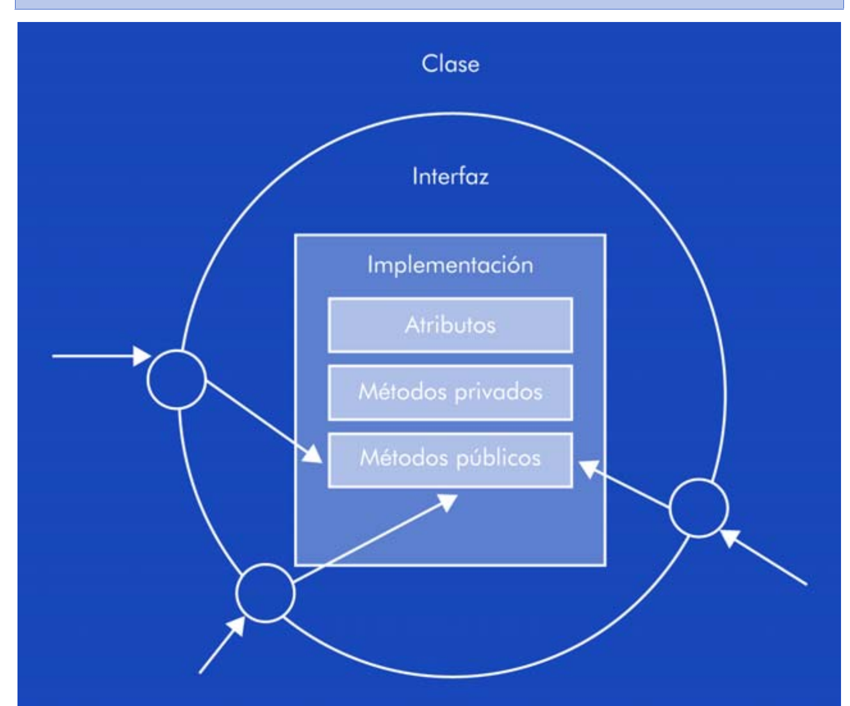
Los programadores consumidores de clases buscarán disponer de unas herramientas en forma de objetos para desarrollar rápidamente nuevas funcionalidades o nuevas aplicaciones a partir de nuestras clases, mientras que el objetivo de los creadores de clases será

ANOTACIONES

construirlas de forma que sólo sea visible su interfaz a los consumidores de clases y el resto esté oculto.

El motivo es que construyendo las clases de esta forma, el creador de la clase podrá cambiar su implementación sin preocuparse del impacto que estos cambios tienen en los consumidores de la clase, reduciendo así los posibles fallos de integración. Estamos evitando que los programadores consumidores de nuestras clases toquen partes de las mismas que son necesarias para el funcionamiento interno, pero que no forman parte de la interfaz de la clase. Estamos además facilitando su tarea, ya que sólo deberán concentrarse en lo que van a usar sin tener que preocuparse de cómo está hecho.

Figura 2. Representación gráfica de los elementos que componen una clase



La encapsulación es la propiedad de las clases y los mecanismos que tendremos en cada lenguaje de programación que nos permite definir qué métodos o atributos son públicos, privados o tienen algún otros mecanismo de protección. Separando la interfaz pública de la implementación interna (privada) facilitaremos también las tareas de reescritura de procesos internos de la clase (para mejorar su rendimiento, por ejemplo).

ANOTACIONES

2.3.5. Reusando la implementación. Composición

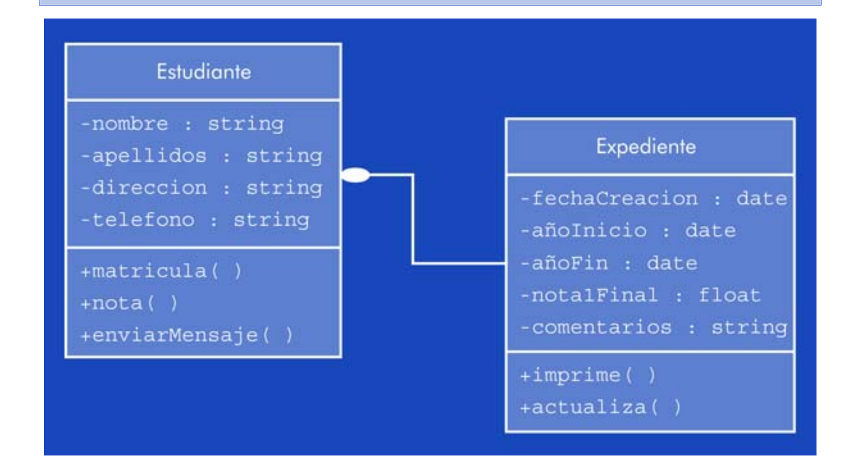
Una vez tenemos una clase desarrollada y probada, debería representar un elemento de código útil en sí mismo. No siempre es así, y se necesita experiencia e intuición para producir clases fácilmente reutilizables. La forma más simple de utilizar una clase es instanciar un objeto de la misma, pero también podemos poner esa clase dentro de otra, creando lo que se llama un "objeto miembro" de la nueva clase. Esta nueva clase puede estar formada por tantos objetos como queramos y en cualquier combinación destinada a solucionar el problema al que nos estamos enfrentando.

Ya que estamos componiendo una nueva clase a partir de otras ya existentes, este concepto se llama "composición" (si la composición ocurre dinámicamente, suele llamarse "agregación"). La composición suele referirse a relaciones del tipo "tiene-un", como por ejemplo "un alumno tiene un expediente".

Nota

Esta figura sigue la notación definida en el *Lenguaje Unificado de Modelado* (UML), en el que la composición se representa mediante el diamante sólido. La agregación se representaría mediante la silueta del diamante.

Figura 3. Representación gráfica de una composición



Típicamente, los objetos miembro de la nueva clase son de acceso privado. Esto permitirá al programador de la misma cambiarlos por otros si es necesario sin cambiar otras partes del código o instanciarlos en tiempo de ejecución para cambiar el comportamiento de la clase.

2.3.6. Reusando la interfaz. Herencia

Una de las más conocidas características de la orientación a objetos es la posibilidad de escoger una clase existente, crear una copia y

ANOTACIONES

después añadir o modificar sus prestaciones en la copia creada. Esto es lo que se entiende por herencia en este paradigma, con la característica adicional de que si la clase original (llamada la clase base, superclase o clase padre) cambia, también cambiará su copia (llamada la clase derivada, heredada, subclase o clase hija).

Figura 4. Representación gráfica de la herencia

UsuarioCampus
-usuario : string
-contraseña : string
+cambiarContraseña( )
+entraCampus( )
+saleCampus( )

Estudiante
-nombre : string
-apellidos : string
-direccion : string
-telefono : string
+matricula( )
+nota( )
+enviarMensaje( )

Nota

La flecha de este diagrama UML apunta de la clase derivada a la superclase.

Cuando heredamos de una clase base, estamos creando un nuevo tipo. Este nuevo tipo no sólo tiene todos los atributos y métodos de la clase padre (aunque los que eran ocultos seguirán siéndolo), sino que duplica completamente su interfaz. Es decir, todos los mensajes que podíamos mandar a un “UsuarioCampus”, los podemos mandar a un “Estudiante”, y como la clase viene determinada por la interfaz, deducimos que un “Estudiante” es también un “UsuarioCampus”.

Ahora bien, la clase “Estudiante” se ha diferenciado de la clase “UsuarioCampus”, ha añadido nuevas propiedades y métodos que “UsuarioCampus” no tenía, en otras palabras, hemos “extendido” la clase “UsuarioCampus”.

En diseño orientado a objetos, siempre debemos estar dispuestos a reestructurar nuestras clases, y precisamente gracias a las propiedades de encapsulación y abstracción, este rediseño no comportará muchos cambios en el código. Veamos la siguiente situación:

Figura 5. Representación gráfica de dos clases heredadas

UsuarioCampus
-usuario : string
-contraseña : string
+cambiarContraseña( )
+entraCampus( )
+saleCampus( )

Consultor
-nombre
-apellidos
-direccion
-telefono
+enviarMensaje( )

Estudiante
-nombre : string
-apellidos : string
-direccion : string
-telefono : string
+matricula( )
+nota( )
+enviarMensaje( )

A primera vista, vemos que todas las clases que heredan de "UsuarioCampus" comparten algunos atributos y métodos, además, conceptualmente, estos atributos también tienen sentido para un "UsuarioCampus". Así pues, lo más natural sería situar estos atributos y métodos en la superclase.

Figura 6. Representación gráfica de dos clases heredadas después de su reestructuración

Ahora nos podríamos plantear si tiene sentido mantener la clase "Consultor" o bien ésta ya no es necesaria. Y aquí es donde aparece la otra forma de diferenciación de la superclase. Para un "Consultor", los métodos `entraCampus()` y `saleCampus()` realizarán operaciones diferentes que para un "Estudiante" (quedará registrado su acceso en estadísticas diferentes, etc.) por lo que en lugar de añadir funcionalidad de "UsuarioCampus" lo que hace "Consultor" es modificar funcionalidad existente (sobreescribir una funcionalidad).

Por lo que hemos visto hasta ahora, la herencia nos permite crear relaciones del tipo "es-un" y del tipo "es-como" según estemos sobreescribiendo o extendiendo funcionalidades respectivamente.

2.3.7. Polimorfismo

Al trabajar con jerarquías de clases de este tipo, nos daremos cuenta de que a veces vamos a querer trabajar con la clase derivada como si fuese la superclase. Esto nos permitirá escribir código que hable con "UsuarioCampus" y afecte a "Estudiante" o "Consultor" sin tener que escribir código para cada uno.

Si más adelante extendemos aún más la clase “UsuarioCampus”, tampoco tendremos que cambiar nuestro código. Esto provoca lo que se llama en términos de compilación el ***late binding***. El compilador no sabe a qué clase estaremos llamando en tiempo de ejecución, sólo puede comprobar que los parámetros de la llamada sean correctos, pero no sabe exactamente qué código ejecutará.

Ya hemos comentado que la implementación del método `entraCampus()` será diferente para “Consultor” y para “Estudiante”, pero el objeto que llame a este método lo hará sobre un objeto de la clase “UsuarioCampus” (ya que tanto “Consultor” como “Estudiante” lo son).

Supongamos que tenemos un código que trabaja con UsuarioCampus, de esta forma:

```
void trabajaUsuarios(UsuarioCampus uc) {
   uc.entraCampus();
   ...
   ...
   uc.cambiarContraseña();
   uc.saleCampus();
}
```

Este código habla con cualquier “UsuarioCampus”, por lo tanto, es independiente del tipo específico de usuario. Si en otra parte del programa usamos este método,

```
Consultor co = new Consultor();
Estudiante es = new Estudiante();
trabajaUsuarios(co);
trabajaUsuarios(es);
```

estas llamadas son perfectamente lógicas. Lo que está sucediendo es que un “Consultor” se está pasando como parámetro a un método que espera un “UsuarioCampus”, pero como un “Consultor” es un “UsuarioCampus”, puede ser tratado como tal por `trabajaUsuarios()`.

ANOTACIONES

El proceso de tratar a una clase derivada como la clase base suele llamarse *upcasting*, por el significado de hacer un *cast* (cambiar el tipo de una variable o objeto) hacia arriba en la jerarquía de objetos:

Figura 7. Camino que sigue la conversión de clases en un *upcasting*

De la misma forma, vemos que el código de la función `trabajaUsuarios()` no contempla todos los tipos de "UsuarioCampus", sino que llama directamente al método e internamente sucede lo correcto. Tanto si "Consultor" o "Estudiante" han sobreescrito ese método en particular como si no, se llama al método apropiado. Esto es polimorfismo en el sentido de que las clases pueden actuar tanto como sus derivadas o como sus clases base según las circunstancias en que se encuentren o los mensajes que reciban.

2.3.8. Superclases abstractas e interfaces

En algunas ocasiones, nos puede interesar que la superclase represente únicamente una interfaz para sus clases derivadas. Es decir, no queremos que nadie instancie un objeto de la superclase, sólo necesitaremos que hagan *upcasting* para usar su interfaz. Esto es lo que se llama definir una clase "abstracta". De la misma forma, un método definido en una superclase puede ser abstracto en el sentido de

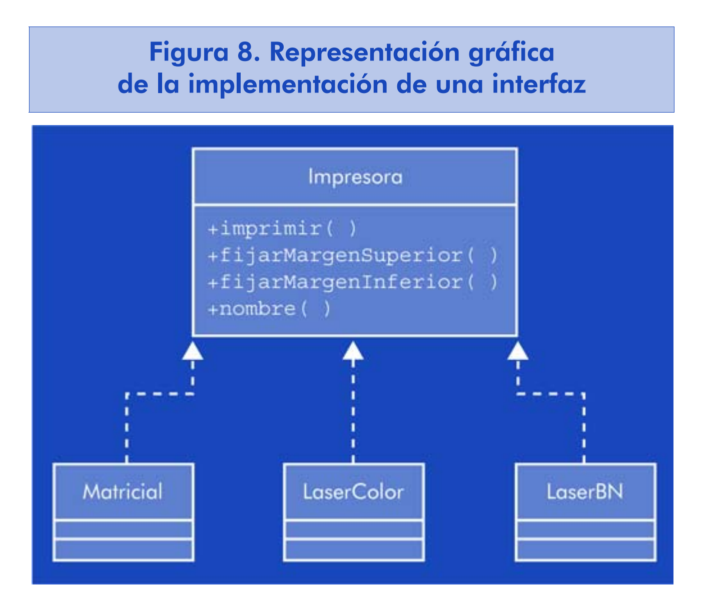
que esa superclase no lo implemente y deberemos sobreescribirlo añadiéndole código en cada clase derivada.

Si una clase tiene métodos abstractos, ya es una clase abstracta. Si todos los métodos son abstractos, entonces habremos separado totalmente la implementación de la interfaz y tendremos una superclase "interfaz". Pensemos por ejemplo en una clase que represente a las impresoras de nuestro sistema. Esta clase tendrá métodos para imprimir, modificar los márgenes, etc. pero no tendrá implementación. Para cada impresora de nuestro sistema deberemos crear una clase hija que implemente estos métodos en cada modelo de impresora.

Nota

La flecha discontinua de este diagrama UML apunta desde la clase que im-plementa la interfaz hacia la clase definida como interfaz.

Figura 8. Representación gráfica de la implementación de una interfaz



2.3.9. La orientación a objetos y la notación UML

En paralelo a los avances en paradigmas de programación, también han evolucionado los métodos y notaciones para modelar y diseñar los sistemas antes de implementarlos. La orientación a objetos no es una excepción y la forma de presentar la solución al problema planteado en forma de objetos que representan las entidades involucradas es muy susceptible de ser modelada de forma visual, es decir, mediante diagramas muy fácilmente entendibles.

El Dr. James Rumbaugh fue uno de los pioneros en técnicas de modelado de clases, jerarquías y modelos funcionales, que según él "capturaban las partes esenciales de un sistema".

ANOTACIONES

El modelado visual de un sistema nos permitirá lo siguiente:

1) Identificar y capturar los procesos de negocio.

2) Disponer de una herramienta de comunicación entre los analistas de la aplicación y los conocedores de las reglas de negocio.

3) Expresar la complejidad de un sistema de forma entendible.

4) Definir la arquitectura del software, sus componentes implicados (interfaz de usuario, servidor de bases de datos, lógica de negocio) independientemente del lenguaje de implementación que usemos.

5) Promover la reutilización, al identificar más fácilmente los sistemas implicados y los componentes.

Así pues, a partir de técnicas desarrolladas por James Rumbaugh en modelado de objetos, Ivar Jacobson en casos de uso, Grady Booch en su metodología de descripción de objetos y con la participación de empresas como HP, IBM, Oracle y Microsoft entre otras, se creó la notación UML bajo el patrocinio de la empresa Rational (recientemente adquirida por IBM) y fue aprobada por la OMG (Object Management Group) en 1997.

Las siglas UML son la abreviatura de *Unified Modeling Language* (Lenguaje Unificado de Modelado) y combina en una sola notación los siguientes modelos:

- Modelado de datos (similar a un diagrama entidad relación).
- Modelado de reglas de negocio y flujos de trabajo.
- Modelado de objetos.
- Modelado de componentes de un sistema.

En pocos años se ha convertido en el estándar para visualizar, especificar, construir y documentar los elementos que intervienen en un sistema software de cualquier tamaño. Puede usarse en cualquier proceso, durante todo el ciclo de vida del proyecto e independientemente de la implementación.

ANOTACIONES

Hay que tener en cuenta que UML no es una metodología, es simplemente una notación para modelar nuestro sistema. Por ello, tampoco está pensado para describir la documentación de usuario, ni siquiera su interfaz gráfica. Así pues, al empezar un proyecto software, deberemos primero escoger la metodología bajo la que vamos a trabajar con él, para después usar UML a lo largo del ciclo de vida que nos marque la metodología escogida. UML tampoco recomienda ninguna en concreto, aunque quizá el método iterativo visto en el capítulo anterior es el más usado y el que mejor se adapta a los proyectos diseñados con el paradigma de orientación a objetos y modelados en UML.

UML ha sido y seguirá siendo el estándar de modelado orientado a objetos de los próximos años, tanto por la aprobación de los expertos en metodologías e ingeniería del software, como por la participación de las grandes empresas de software, la aceptación del OMG como notación estándar y la cantidad de herramientas de modelado que lo soportan.

2.3.10. Introducción a UML

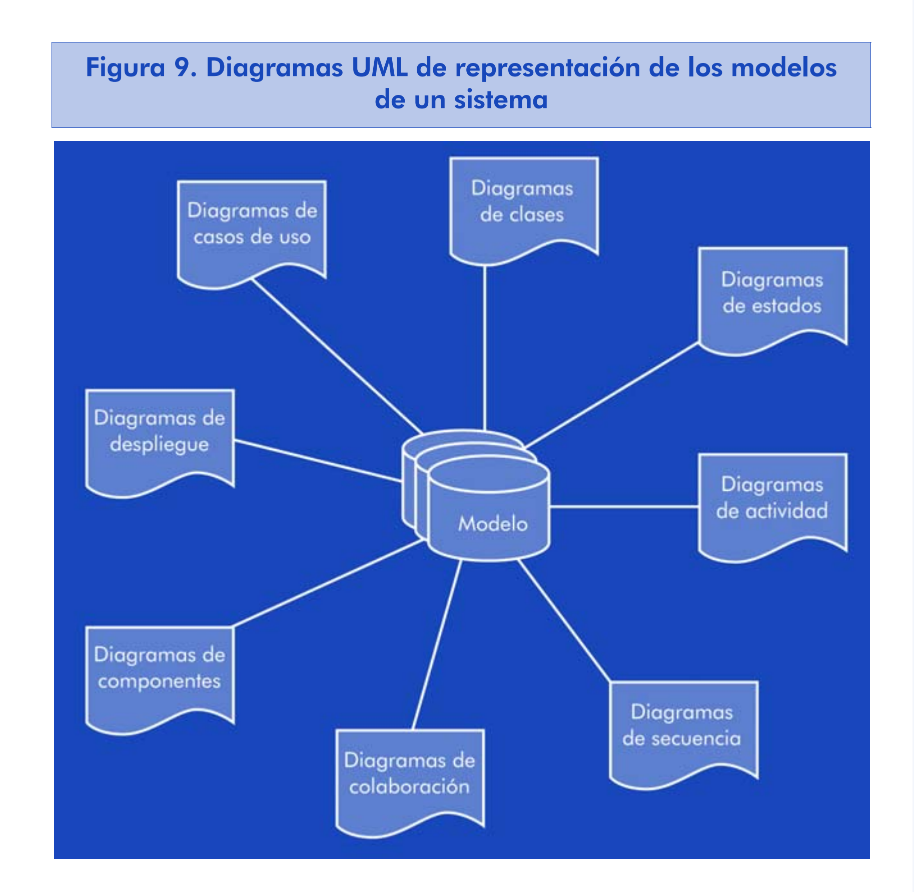
Un modelo es una abstracción de un sistema o de un problema que hay que resolver, considerando un cierto propósito o un punto de vista determinado. El modelo debe describir completamente los aspectos del sistema que son relevantes a su propósito y bajo un determinado nivel de detalle.

El código fuente es también una expresión del modelo, la más detallada y la que además implementa la funcionalidad del mismo, pero no es cómodo como herramienta de comunicación. Además, para llegar a él es conveniente desarrollar antes otras representaciones.

Un diagrama nos permitirá representar gráficamente un conjunto de elementos del modelo, a veces como un grafo con vértices conectados, y otras veces como secuencias de figuras conectadas que representen un flujo de trabajo.

ANOTACIONES

Figura 9. Diagramas UML de representación de los modelos de un sistema



Cada punto de vista del sistema (y cada nivel de detalle) podrá modelarse y ese modelo podrá representarse gráficamente. Lo que UML propone es una notación y un conjunto de diagramas que abarcan las perspectivas más relevantes del sistema.

- Diagrama de casos de uso
- Diagrama de clases
- Diagramas de comportamiento
  - Diagrama de estados
  - Diagrama de actividad
  - Diagramas de interacción
    - Diagrama de secuencia
    - Diagrama de colaboración
- Diagramas de implementación
  - Diagrama de componentes
  - Diagrama de despliegue

Estos diagramas responden a las vistas de un sistema software. Desde la definición del problema (casos de uso), la vista lógica (clases, objetos), la vista de procesos (comportamiento) y la vista de implementación y distribución.



La última especificación de UML disponible es la revisión 1.5 y la versión 2.0 está en su última fase de revisión que terminará durante el año 2005 e incorpora conceptos como la descripción de infraestructuras, el intercambio de diagramas y la expresión de restricciones en objetos (*constraints*).

Existen multitud de herramientas de soporte a la creación de diagramas UML. A lo largo de los siguientes capítulos mostraremos las más representativas con licencia de código abierto y los diagramas que presentaremos estarán hechos con esas herramientas para mostrar mejor al estudiante sus posibilidades y diferencias. Las herramientas que utilizaremos son Dia, Umbrello y ArgoUML.

2.3.11. Conclusiones

En este apartado hemos hecho un rápido repaso por los conceptos relacionados con el paradigma de la orientación a objetos. Hemos intentado describir sus características más importantes y proporcionar ejemplos ilustrativos, a la vez que hemos introducido algunos elementos de la notación UML que veremos en detalle en apartados posteriores.

Lectura recomendada

Para profundizar en este paradigma o adquirir experiencia en programación orientada a objeto os recomendamos que consultéis los textos siguientes:

B. Eckel (2003). Thinking in Java (3.º ed.). Upper Saddle River: Prentice Hall. http://www.mindview.net/Books/TIJ/

Java Technology. http://java.sun.com

E. Gamma; R. Helm y otros (1995). Design Patterns. Reading, Mass: Addison Wesley.

Object Management Group. http://www.omg.org/

También hemos introducido la notación UML, su historia y sus características principales.

2.4. Definición del caso práctico

Para entender una notación de modelado como UML, podríamos simplemente leer la especificación que publica la OMG y tendríamos la visión más completa posible del mismo. Pero tratándose de un estándar que indica cómo debemos modelar un sistema, es mucho más cómodo y fácil de entender haciéndolo a través de un caso práctico en el que podamos aplicar cada concepto a un caso concreto, a la visión que tienen del mismo sus participantes, y reflejar en un diagrama cada una de estas visiones y casos.

ANOTACIONES

El caso práctico que planteamos es el siguiente:

Una empresa de desarrollo de software tiene problemas para gestionar sus operaciones con los clientes, tanto si se trata de futuros clientes a los que ha presentado una propuesta (o está pendiente de hacerlo), como si se trata de clientes que han contratado un proyecto y hay que hacer su seguimiento, no sólo en términos de tareas pendientes, horas invertidas, etc. sino también en términos de facturación, importes pendientes de pago, etc.

Para ello, podría decidirse por implantar un ERP basado en software libre y parametrizarlo según sus necesidades, pero después de evaluar algunas de las alternativas existentes, optó por desarrollar un sistema completo de gestión de proyectos y clientes a la medida de su método de trabajo. Además, si el sistema es lo suficientemente flexible, podrá liberarlo bajo alguna licencia aprobada por la OSI y obtener beneficios prestando soporte, formación y actualizaciones del mismo.

Es obvio que no vamos a describir aquí el proyecto, ya que ésta es la tarea que vamos a realizar mediante UML en los apartados posteriores. Simplemente vamos a enumerar algunas de sus características más representativas para tener conceptos y casos con los que empezar a trabajar.

El sistema de gestión deberá permitir:

1) Disponer de una lista completa de contactos de la empresa. Sean éstos clientes, posibles clientes o proveedores.

2) Disponer de una lista completa de proyectos en los que está trabajando la empresa. De cada proyecto deberemos disponer de información sobre:

- El cliente al que hace referencia.
- El estado actual del proyecto.
- La estimación que se hizo en su momento (el presupuesto que se presentó al cliente).
- Las tareas que implica el proyecto, su estado, duración estimada y real.

ANOTACIONES

- Los costes que ha tenido el desarrollo del proyecto.

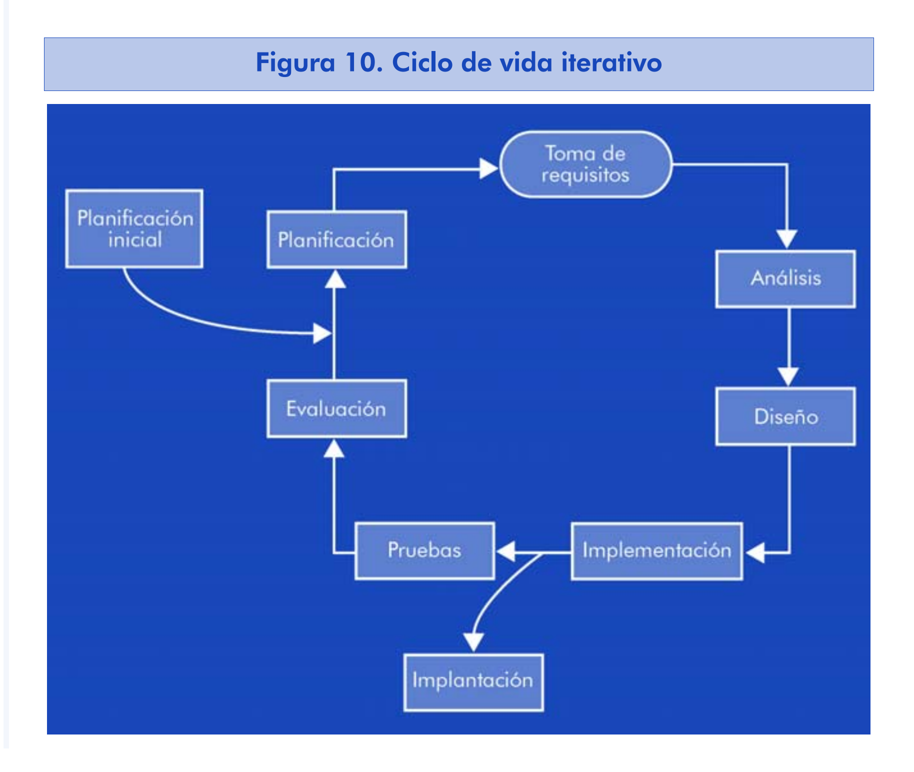
3) Disponer de una lista de desarrolladores que podamos asociar a cada proyecto. Éstos deberán registrar las horas trabajadas en cada tarea, sus avances, etc.

4) Disponer de informes en los que podamos ver qué proyectos se han retrasado, cuál ha sido el resultado de un proyecto respecto a su planificación inicial, etc.

Con estos datos, damos por planteado el caso de estudio. No entra dentro de los objetivos de este capítulo su resolución, por lo que la especificación no es todo lo completa que debería ser en la realidad, aunque suficiente para empezar a trabajar.

Tal como hemos comentado en apartados anteriores, es fundamental determinar la metodología que vamos a usar antes de empezar el proceso de modelado. En nuestro caso, al ser una empresa de desarrollo de software moderna y con una clara inclinación hacia el software libre, nuestro método de desarrollo y gestión de proyectos se ajusta perfectamente al que caracteriza un ciclo de vida de tipo iterativo.

Figura 10. Ciclo de vida iterativo



ANOTACIONES

Aunque nos centraremos en las fases de toma de requisitos, análisis y diseño, algunos de los diagramas se basarán en modelos de las fases de implementación e implantación. No es posible establecer una secuencia perfecta entre los diagramas, ni una correspondencia total con las diferentes fases del ciclo de vida, ya que dependiendo de las personas involucradas en cada fase y de su experiencia y conocimiento, algunos diagramas pueden quedar completados en fases muy tempranas mientras que otros van a estar sometidos a continuas revisiones. Teniendo en cuenta esto, una posible correspondencia es la siguiente:

Figura 11. Correspondencia de los diagramas UML con fases de un proyecto

Aunque la figura muestra algunos de los diagramas unidos por flechas, casi todos ellos tienen relación con los demás, de forma que un cambio en alguno puede afectar a muchos otros. Sólo hemos indicado las correspondencias más evidentes, aunque es de esperar que un cambio en un caso de uso, por ejemplo, conllevará cambios en muchos otros diagramas, o que un cambio en el diagrama de componentes puede afectar a los diagramas de clases y de despliegue.

Como la lectura del material nos impone un orden, vamos a seguir, en la medida de lo posible, el mismo de la figura, de arriba abajo y de izquierda a derecha, avanzando por las fases del proyecto.

ANOTACIONES

2.5. Diagramas de casos de uso

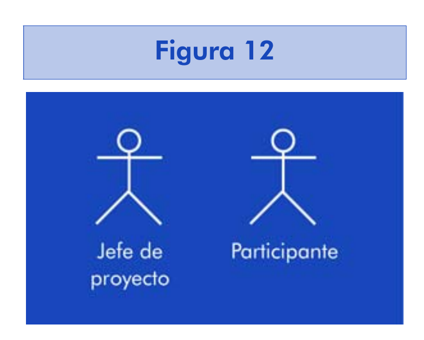
Los casos de uso son una herramienta esencial en la toma de requisitos del sistema. Nos permiten expresar gráficamente las relaciones entre los diferentes usos del mismo y sus participantes o actores. El resultado es un conjunto de diagramas muy fácilmente entendibles tanto por el cliente, como por los analistas del proyecto.

No definen todos los requisitos (por ejemplo, tipos de datos, interfaces externas, rendimiento, etc.) pero sí que representan el hilo conductor que los vincula a todos con los actores del sistema.

Se componen de los siguientes elementos:

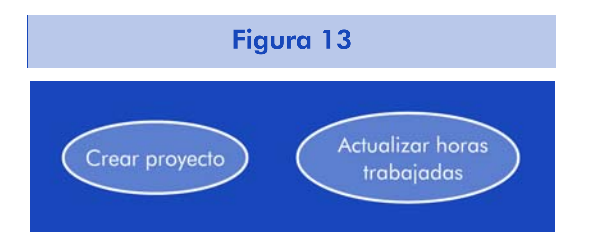
- **Actores**: representan los roles que juegan los usuarios u otros sistemas en el sistema del problema. Identificar a los actores de un caso de uso pasa por averiguar quién está involucrado en cada requisito concreto, quién se beneficiará de cada funcionalidad o quién proveerá o usará la información.

Figura 12



- **Caso de uso**: son las acciones que pueden tener lugar en el sistema que queremos modelar. Para identificarlas, puede ser útil preguntarse cuáles son las tareas y responsabilidades de cada actor, si habrá actores que recibirán información del sistema, etc.

Figura 13

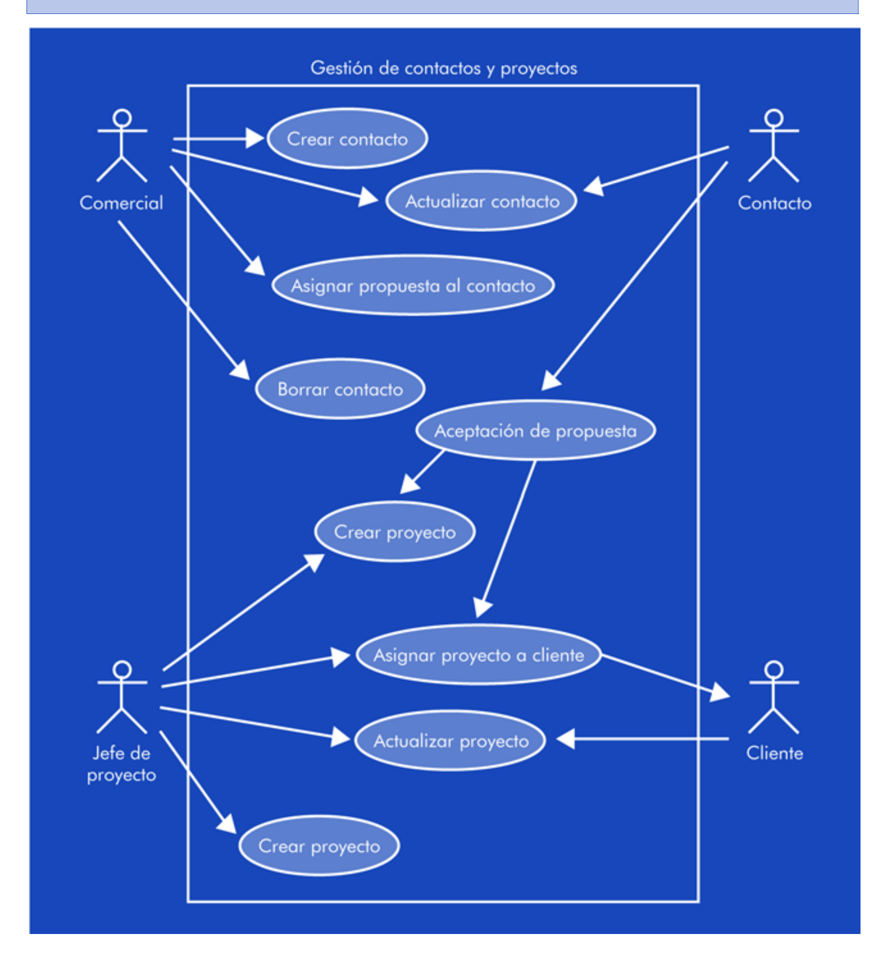


- **Relaciones**: indican actividad o flujo de información.

- **Límite del sistema**: define el ámbito donde se produce el caso de uso que estamos representando y que va a ser tratado por el sistema. Los actores no son parte del sistema y por lo tanto están fuera de sus límites.

A continuación veamos una propuesta de caso de uso en el ámbito de la gestión de los contactos y proyectos.

Figura 14. Caso de uso para la gestión de contactos y proyectos. Realizado con Umbrello



En primer lugar, vemos que estamos modelando a muy alto nivel, trasladando lo expresado en los requerimientos a una representación gráfica. Las relaciones entre los actores y los casos de uso indican si proporcionan o reciben información (según el sentido de la flecha).

ANOTACIONES

En cambio, las relaciones entre casos de uso pueden tener significados diferentes. Cada caso de uso es susceptible de representarse más detalladamente en otro diagrama. Ampliemos uno de ellos para mostrarlo.

Figura 15. Caso de uso para la aceptación de la propuesta

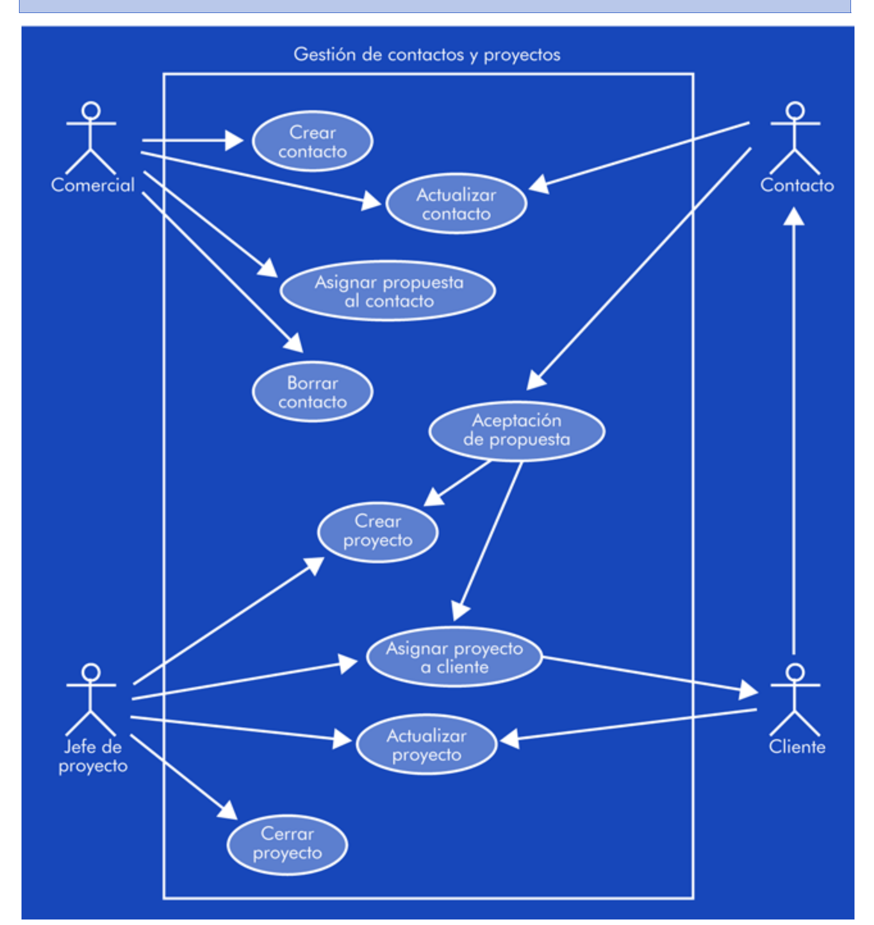
La etiqueta <<incluye>> entre dos casos de uso indica que uno tiene la funcionalidad de otro como parte integrante suya. En el ejemplo, la aceptación de la propuesta incluirá la funcionalidad para actualizar los datos del cliente (ahora tenemos un cliente en lugar de un simple contacto), emitir la factura inicial, quizá (no forzosamente) crear el proyecto, etc.

La etiqueta <<extiende>> indicará que un caso de uso amplía la funcionalidad de otro. En el ejemplo, la emisión de la factura inicial no es la emisión de una factura cualquiera, implicará comprobaciones, como por ejemplo si tenemos todos los datos del cliente, etc., que no se producen en la emisión de una factura normal. Ahora bien, está claro que habrá una parte de funcionalidad que compartirán. En otras palabras, la creación de la factura inicial es un caso particular de la emisión de una factura.

La etiqueta <<generaliza>> indica también una relación padre-hijo similar a la extensión, con la diferencia de que actúan exactamente igual en cuanto a reglas de negocio se refiere. En otras palabras, en un momento dado podríamos sustituir el caso de uso padre por el hijo y el sistema no se vería afectado. En cambio, en una relación de extensión esto no sucede, no podemos sustituir la emisión de cualquier factura por la de la factura inicial.

Las relaciones de generalización también pueden darse entre los actores. Volviendo al diagrama inicial:

Figura 16. Caso de uso para la gestión de contactos y proyectos, con generalización en los actores



Vemos que generalizamos a un cliente en un contacto; es completamente lógico si consideramos que todos los clientes fueron contactos en un cierto instante y que deben poder mantener sus mismas funcionalidades.

ANOTACIONES

Dado el alto nivel al que estamos modelando el sistema en esta fase, es importante tener presentes algunas buenas prácticas con respecto a la interpretación y creación de diagramas de casos de uso:

- Empezar los nombres de los casos de uso con un verbo.
- Aunque no hay forma de indicar el orden en que un actor aplicará los casos de uso, suele ser más intuitivo representarlos en orden descendiente, situando los más importantes en la parte superior del diagrama.
- Situar a los actores principales en la parte superior del diagrama también ayuda a su comprensión y a generalizar de forma más entendible.
- Nombrar a los actores con sustantivos relacionados con las reglas de negocio, de acuerdo con los roles que representan. No con su cargo o posición en el sistema.
- Anteponer "Sistema" o <<sistema>> a los actores que sean procesos externos.
- Para representar eventos que ocurren de forma programada, podemos introducir un actor de sistema "Tiempo" para modelar sus casos de uso.
- La etiqueta <<incluye>> no es obligatoria. Es aconsejable incluirla sólo si en un punto específico la lógica del caso de uso incluido es necesaria.
- No debe abusarse de la etiqueta <<extiende>>, ya que dificulta la comprensión del caso.
- La generalización de clases suele identificarse porque una sola condición del caso de uso (en el ejemplo, el método de envío), cambia por completo su lógica interna, pero no así las reglas de negocio del sistema.

ANOTACIONES

- Debe evitarse representar más de dos niveles de asociaciones de casos de uso en un mismo diagrama. Si nos encontramos en esta situación, hay muchas probabilidades de que estemos haciendo una representación funcional de lo que ocurre en el caso de uso. Debemos concentrarnos sólo en los requisitos de uso, ya que la descomposición funcional es parte de la fase de diseño.

- Situar los casos incluidos a la derecha del caso que los incluye ayuda a comprender mejor el diagrama. De la misma forma, es más intuitivo situar los casos que extienden debajo del caso padre, al igual que los casos que heredan o generalizan.

- Es útil intentar expresar con "es como" la generalización de actores para comprobar si los estamos modelando correctamente.

2.6. Diagramas de secuencia

Los diagramas de secuencia modelan el flujo de la lógica dentro del sistema de forma visual, permitiendo documentarla y validarla. Pueden usarse tanto en análisis como en diseño, proporcionando una buena base para identificar el comportamiento del sistema.

Típicamente se usan para modelar los escenarios de uso del sistema, describiendo de qué formas puede usarse. La secuencia puede expresar tanto un caso de uso completo como quizá un caso concreto del mismo contemplando algunas alternativas.

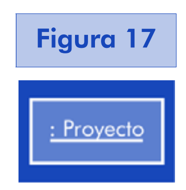
También son una buena herramienta para explorar la lógica de una operación compleja o los elementos implicados en la prestación de un servicio. Nos pueden ayudar a identificar cuellos de botella en la fase de diseño, detectar cuáles van a ser las clases más complejas de implementar y decidir cuáles de ellas van a necesitar diagramas de estados (que veremos más adelante) para facilitar su implementación.

Se componen de los siguientes elementos:

- **Objeto**: instancia de una clase que podemos empezar a identificar como participante en la secuencia de operaciones que representa este caso de uso.

ANOTACIONES

Figura 17



- **Actor**: los actores pueden comunicarse con los objetos, por lo tanto formarán parte de este diagrama.

- **Vida del objeto**: indicamos la existencia de un objeto a lo largo del tiempo mediante una línea discontinua. El fin del mismo se indica mediante un aspa.

- **Activación**: indicamos cuándo el objeto está realizando una tarea concreta.

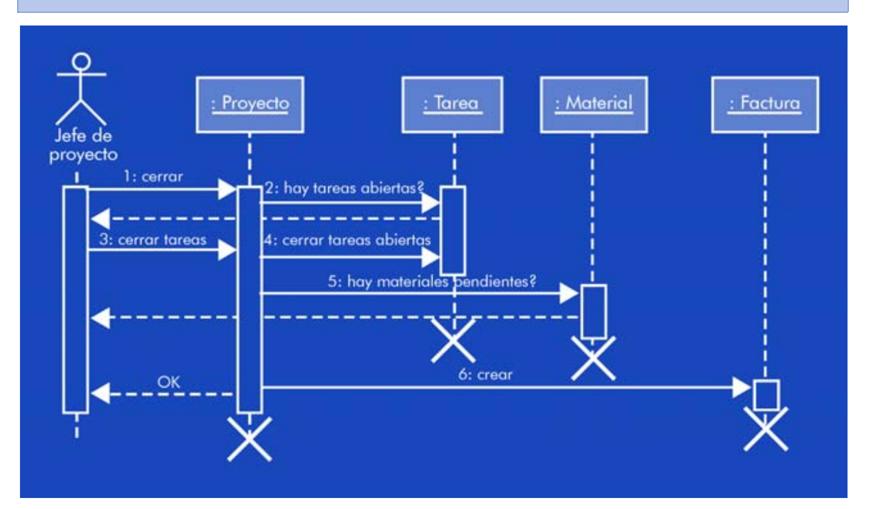
Figura 18



- **Mensaje**: la comunicación entre objetos y sus activaciones.

A continuación, veamos un ejemplo de diagrama de secuencia para el caso de uso implicado en cerrar un proyecto:

Figura 19. Diagrama de secuencia para el caso de uso "Cerrar un proyecto". Realizado con Dia



ANOTACIONES

Una de las características de los diagramas de secuencia es que casi no necesitan aclaraciones en cuanto a su notación. En la parte superior, vemos los participantes en la secuencia (también denominados "clasificadores" que implementan la misma). En general siempre será un actor el que inicie la secuencia, y el resto de participantes pueden ser objetos (representados con una caja en la que escribiremos el nombre del objeto), otros actores, o quizá un caso de uso completo.

Cada participante tiene un intervalo en el que éste está activo en la secuencia, que se indica mediante un rectángulo sobre la línea discontinua que representa la vida del objeto. El rectángulo empieza cuando el objeto recibe un mensaje (representado mediante una flecha que incorpora el nombre de la llamada) y termina cuando éste devuelve su última respuesta (representada mediante una flecha discontinua).

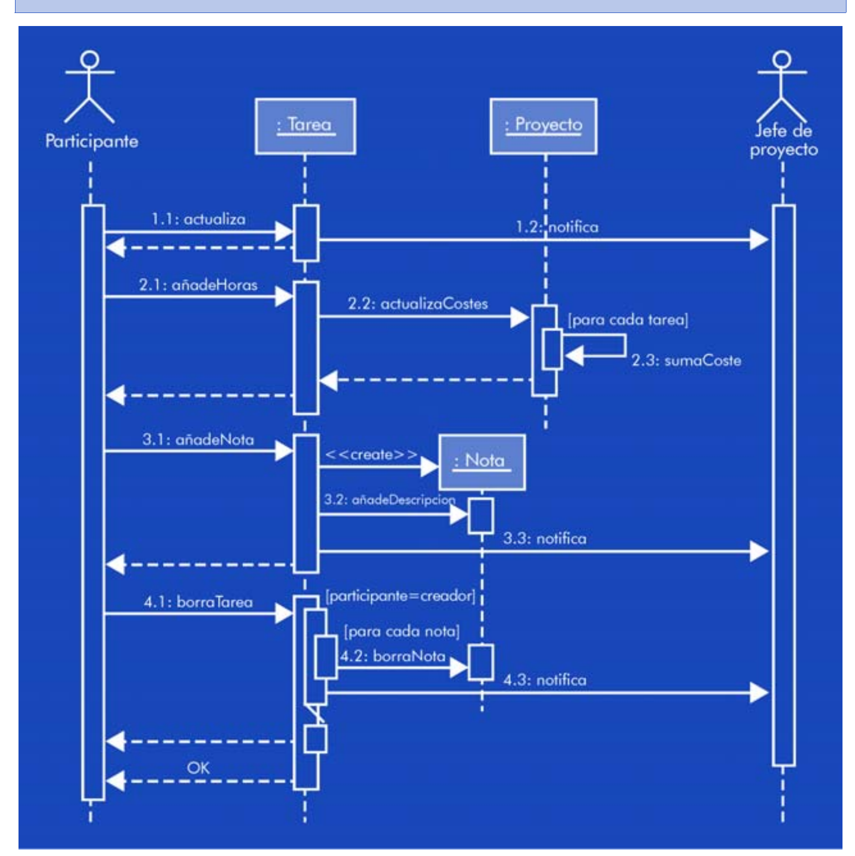
Los mensajes suelen incluir números de secuencia que facilitan la comprensión del diagrama y el seguimiento del orden en que se producen los mensajes. A veces un error puede provocar una línea discontinua de retorno hasta el primer participante, de forma similar a cómo se propagaría una excepción. Así pues, las líneas de retorno pueden incluir también etiquetas para indicar si representan un error o no.

Finalmente, un aspa al final de la vida del objeto indica que éste puede destruirse.

Según la notación, es posible expresar iteraciones y condicionales en un diagrama de secuencia. Muchos expertos no lo recomiendan, ya que implica incluir lógica dentro de un diagrama que sólo debería representar mensajes entre participantes en la secuencia, pero a veces es imprescindible hacerlo para reflejar correctamente la secuencia de mensajes. Veamos otro ejemplo a partir del caso de uso relacionado con la actualización de las tareas en la gestión de proyectos.

ANOTACIONES

Figura 20. Diagrama de secuencia correspondiente a la gestión de un proyecto, con iteraciones y condiciones

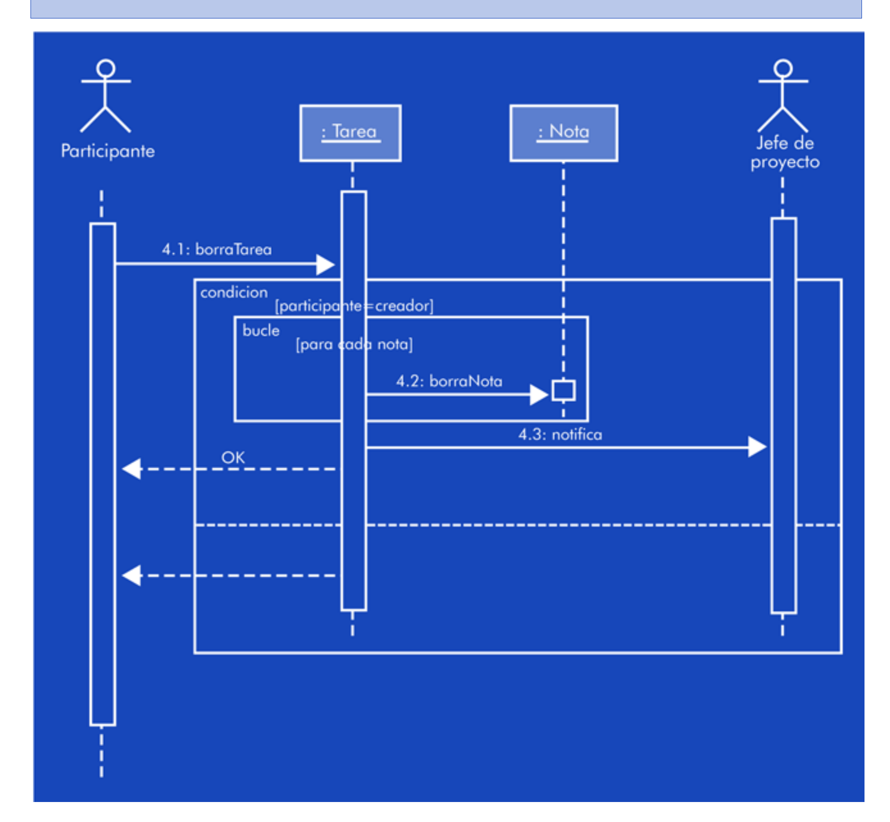


Las guardas de las condiciones o de los bucles se expresan entre corchetes. Para indicar que estamos actuando sobre el mismo objeto, pintaremos un rectángulo ligeramente desplazado a la derecha. En general, los bucles se identificarán por el texto entre corchetes, que indicará sobre qué se está iterando (p. ej. "para cada tarea"), mientras que la guarda de la condición deberá ser todo lo explícita posible.

En caso de necesitar una acción alternativa ("participante!=creador" en el ejemplo), cruzaremos una línea sobre otro rectángulo desplazado para indicar la actividad que tendrá lugar si no se cumple la condición. Puede apreciarse fácilmente que la tendencia a introducir, lógica compleja dentro de los diagramas de secuencia, no ayuda a su esclarecimiento, más bien al contrario. La versión 2.0 de UML introduce el concepto de los "marcos de interacción" que ayudan a mejorar la representación en estos casos:

ANOTACIONES

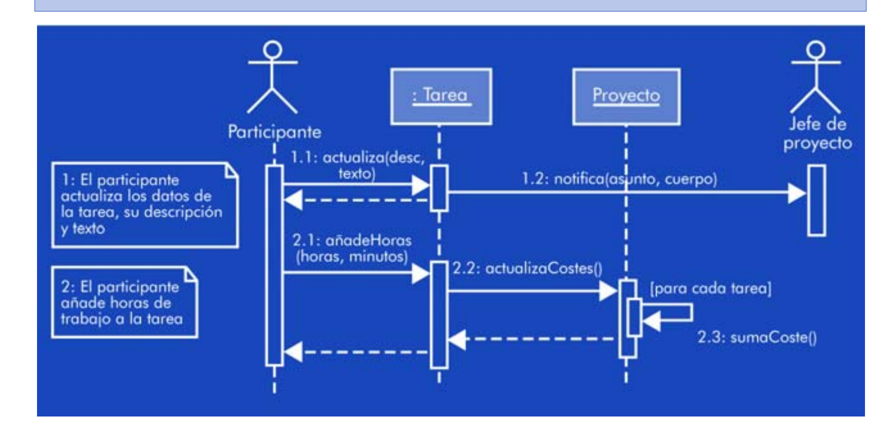
Figura 21. Diagrama de secuencia con iteraciones y condiciones en notación UML2.0



Los marcos de interacción se representan con cajas transparentes que envuelven la condición o bucle. En la esquina superior izquierda se escribe el tipo de interacción (bucle o condición), y en la línea de actividad del objeto, la condición entre corchetes. En el caso de condiciones con alternativa, ésta se separa mediante una línea de puntos.

Cuando este tipo de diagramas se usan en fase de diseño, se suele disponer de más información sobre los objetos y sus mensajes. Si el diagrama se realiza en esa fase, su representación puede ser más completa:

Figura 22. Diagrama de secuencia realizado en fase de diseño



Los mensajes pueden incluir la información que va a necesitar el objeto receptor para realizar su actividad. También hemos incluido unas notas en cada secuencia para indicar de forma textual la actividad que representan.

Finalmente, enumeramos un conjunto de buenas prácticas en la representación de diagramas de secuencia:

- El orden entre los mensajes y los participantes debe ser siempre de izquierda a derecha y de arriba abajo para facilitar la comprensión del diagrama.

- El nombre de los actores debe ser consistente con los casos de uso.

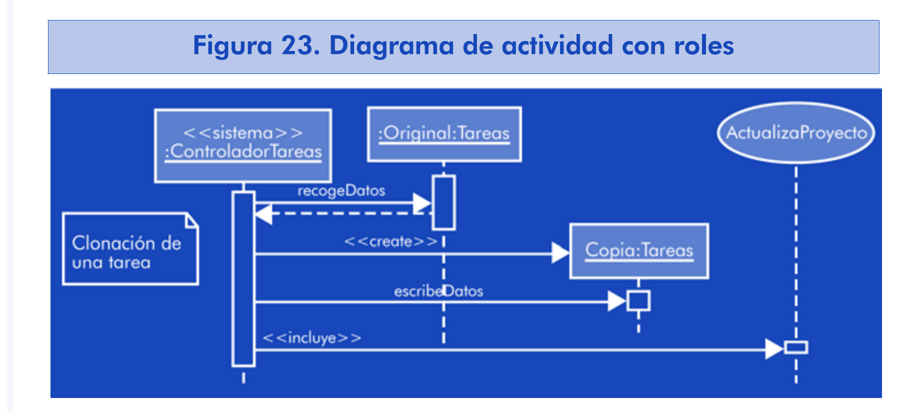
- El nombre de los objetos debe ser consistente con los diagramas de clases.

- Incluir notas en las secuencias.

- Sólo incluir el aspa de destrucción del objeto en casos en que proporcione información sobre cuándo debe destruirse, en caso contrario "ensuciaremos" el diagrama innecesariamente.

- Si la secuencia trabaja con varios objetos de la misma clase pero con roles diferentes, podemos etiquetarlos de la manera siguiente:

Figura 23. Diagrama de actividad con roles



Vemos que tenemos dos instancias del objeto Tarea, una para la original y otra para la copia.

- Usar los estereotipos de forma consistente. En el ejemplo anterior, el sistema controlador de tareas es quien inicia la clonación, y lo indicamos mediante la palabra <<sistema>>.

- En los parámetros en mensajes, es más conveniente usar nombres claros que los tipos de los mismos.

- En las llamadas a casos de uso, usar el estereotipo <<incluye>>.

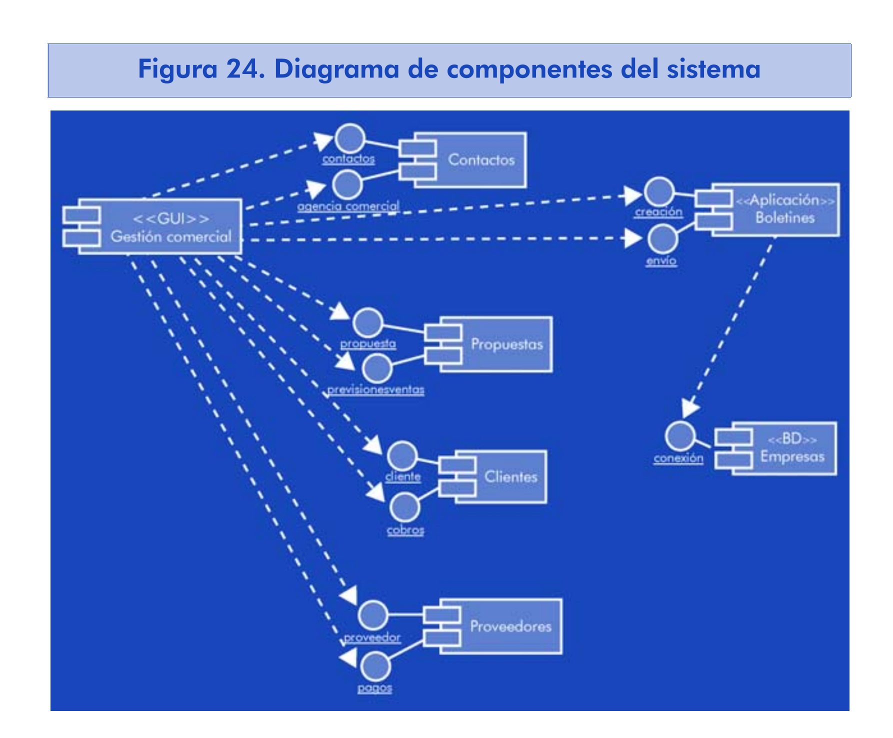
2.7. Diagramas de componentes

El desarrollo orientado a objeto está muy relacionado con el desarrollo basado en componentes, en el sentido de que las clases con sus propiedades de encapsulación y abstracción se ven en muchas ocasiones como componentes cerrados de un sistema. Así pues, la notación UML incluye un diagrama de componentes que según la definición del OMG "muestra las dependencias entre componentes software, incluyendo los clasificadores que los especifican (por ejemplo las clases de implementación) y los artefactos que los implementan, como los ficheros fuente, binarios, scripts, etc.".

En muchos casos en los que podrían usarse diagramas de componentes, se usan los diagramas de despliegue (que veremos más adelante), ya que éstos nos permiten modelar además dónde va a implantarse cada componente y bajo qué configuración o parámetros. Así pues, el diagrama de componentes ha quedado tradicionalmente relegado a modelar la arquitectura del sistema a nivel lógico o de entorno de negocio.

ANOTACIONES

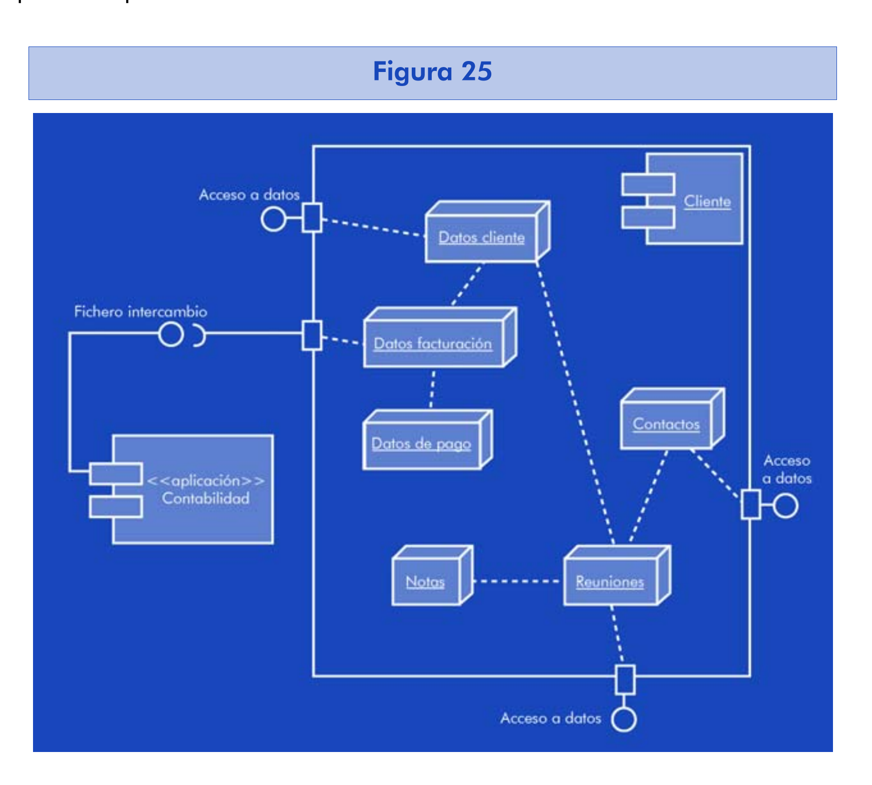
Figura 24. Diagrama de componentes del sistema



Aquí hemos modelado la estructura de los componentes software que están involucrados en la aplicación de gestión comercial. Los componentes incorporan puertos hacia los datos o servicios que proporcionan (en el ejemplo "contacto", "agenda comercial", etc.) o puertos para recibir datos o resultados de peticiones.

Cada puerto puede tener una o más interfaces. Ampliemos un componente para mostrarlo más en detalle:

Figura 25



ANOTACIONES

Hemos desarrollado el componente "Cliente" en sus partes constituyentes (clases, muy probablemente) y hemos definido sus interfaces a alto nivel. Vemos que básicamente ofrece interfaces de acceso a sus datos (de entrada y salida) y una interfaz de exportación de datos de facturación para comunicarse con la aplicación de contabilidad. Aquí hemos utilizado partes de la notación UML2.0, que diferencia entre interfaces que el componente ofrece (círculo cerrado) e interfaces que el componente necesita (círculo abierto).

Los conceptos de *interfaz* y *puerto* serán muy familiares a los estudiantes que tengan experiencia en desarrollo basado en componentes (CORBA, COM, IDL, etc.) pero en la mayoría de los casos no será necesario llegar a este nivel de detalle en el diagrama en aplicaciones orientadas a objeto convencionales.

En otros entornos donde sea habitual utilizar patrones de diseño, los diferentes objetos, interfaces y la comunicación entre ellos se pueden representar en los detalles del componente.

Finalmente, la lista de buenas prácticas en la representación de componentes es la siguiente:

- Aplicar los estereotipos de forma consistente.

- Aunque usemos UML 1.5, la representación de las interfaces de los componentes mediante círculos es mucho más clara que una simple flecha entre ellos.

- Ya que la interfaz es un conjunto de métodos, intentemos no representar demasiadas interfaces, es aconsejable agruparlas bajo un mismo nombre y crear después un diagrama detallado del componente.

- Los componentes pueden heredar unos de otros. En este caso, la flecha que utilizamos en los casos de uso para expresar generalización puede servir perfectamente.

- Para mejorar la comprensión de un diagrama de componentes, es preferible conectarlos siempre a través de interfaces. Es más consistente y evita confusiones o dudas sobre su interpretación.

ANOTACIONES

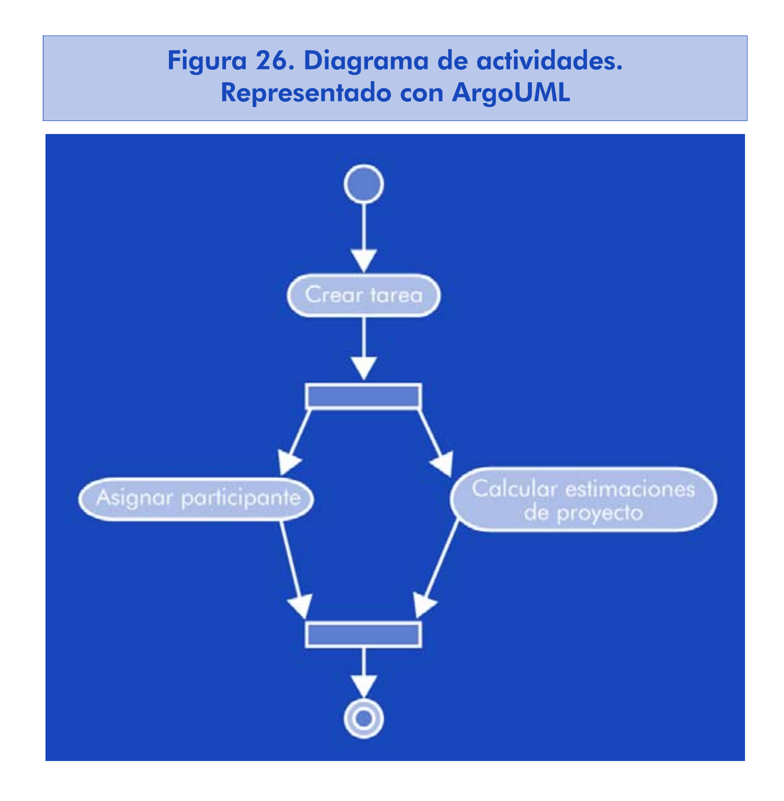
ANOTACIONES

2.8. Diagrama de actividades

En muchos aspectos, los diagramas de actividades son el equivalente orientado a objeto de los diagramas de flujo y los DFD del desarrollo estructurado. Su uso es principalmente la exploración y representación de la lógica comprendida en operaciones complejas, reglas de negocio, casos de uso o procesos software.

De forma similar a los tradicionales diagramas de flujo, todo diagrama de actividades tiene un punto de partida y un final. Las actividades representarán cada paso importante que se produce en el proceso que estamos modelando (puede representar un caso de uso o bien un conjunto de ellos).

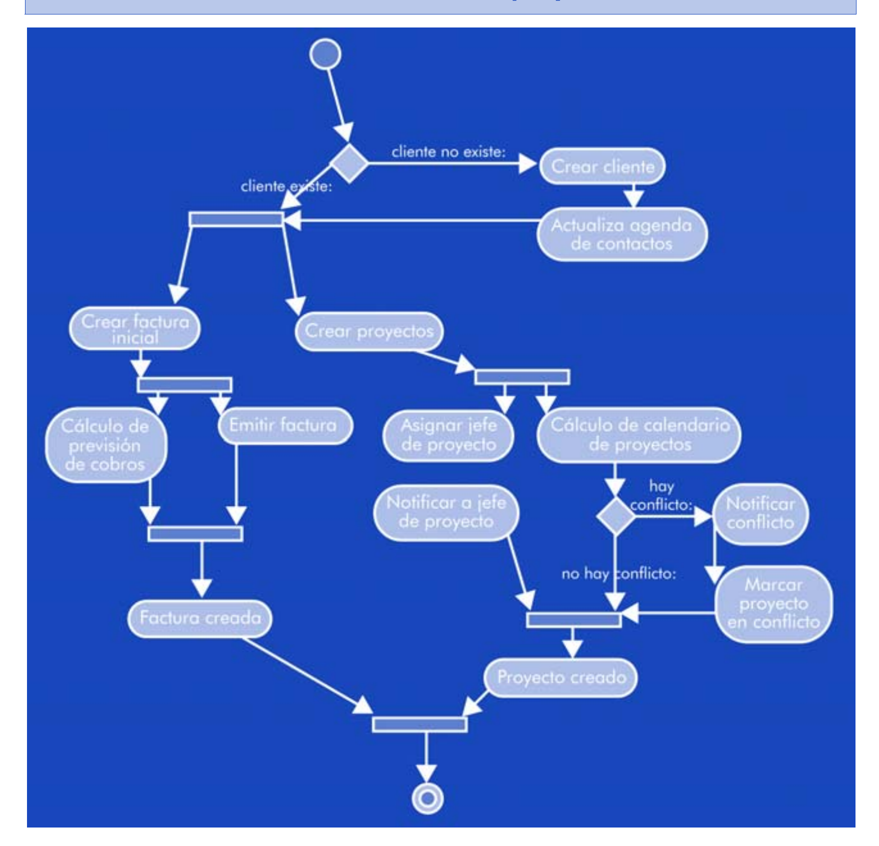
Figura 26. Diagrama de actividades. Representado con ArgoUML



Las transiciones son la representación del flujo de información o proceso que avanza entre actividades. A diferencia de los diagramas de flujo, el diagrama de actividades permite modelar acciones en paralelo. Para dividir el proceso o bien recuperar un único flujo, se utilizan las barras de sincronización que permiten varios flujos de entrada o varios de salida.

Veamos a continuación un ejemplo más completo:

Figura 27. Diagrama de actividades correspondiente a la creación de un proyecto

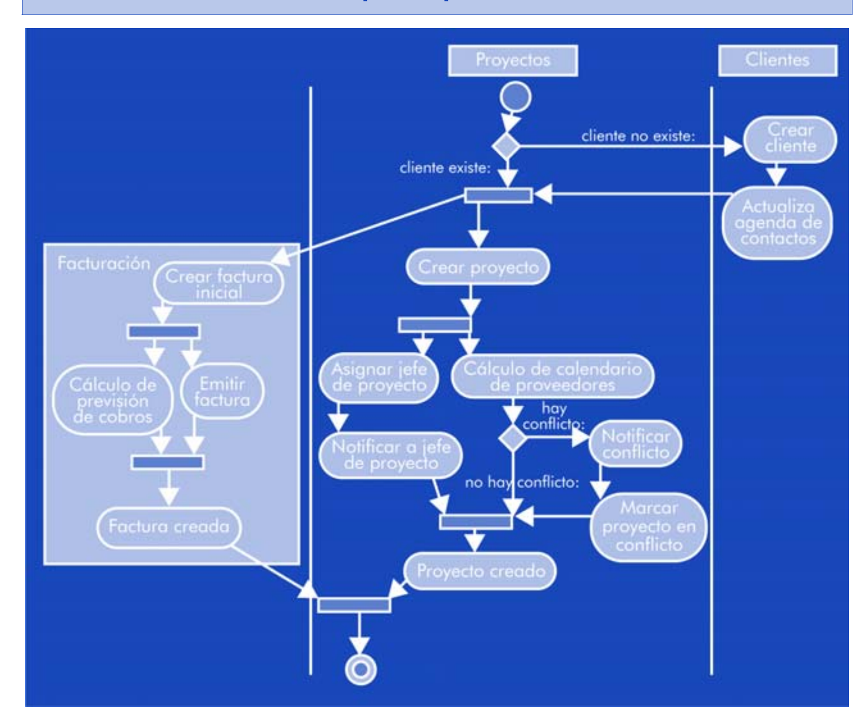


Puede apreciarse que los elementos condicionales son similares a los de los diagramas de flujo: un rombo con las transiciones etiquetadas según sea el resultado de la condición evaluada.

Algunos autores consideran una buena práctica marcar los ámbitos de actuación de clases o componentes en el diagrama, para dar más información sin añadir complejidad. Para ello, pueden usarse sombras bajo los objetos o bien organizar las actividades en columnas etiquetadas.

ANOTACIONES

Figura 28. Diagrama de actividades con separación del ámbito en que se producen



El diagrama de actividades, al incorporar mecanismos de sincronización, permite expresar qué actividades se pueden realizar en paralelo y cuáles deben realizarse en serie. La implementación podría optar por no paralelizar todo lo posible, pero debemos tener presente que estamos representando el modelo, no expresando su implementación.

También es muy importante representar correctamente la sincronización cuando dos o más flujos de actividad convergen en un punto. En el ejemplo hemos preferido expresar dos actividades “Proyecto creado” y “Factura creada” para evitar sincronizar todas las actividades finales en un mismo punto. Es un concepto que clarifica la representación y en el momento de la implementación no va a afectar a la calidad del resultado, ya que si la actividad simplemente devuelve un valor o recoge los resultados de las actividades anteriores, el diagrama seguirá siendo igual de correcto. Si además se realizan tareas de consolidación de datos, liberación de objetos etc., éstas podrán realizarse también en paralelo.

El diagrama de actividades de UML 1.5 no va más allá. En UML 2.0 se añaden un buen número de elementos a su notación para expresar el tiempo, el tipo de actividad (una simple llamada, una transformación

ANOTACIONES

de datos, la transmisión de un objeto y cuál), o la representación de eventos externos que afectan al flujo de actividades.

Finalmente, veamos algunas buenas prácticas relacionadas con estos diagramas:

- Situar el punto de inicio en la parte superior del diagrama.

- Aunque algunos autores lo consideran opcional (una actividad puede ser el final del diagrama), siempre es conveniente situar un punto de final de la actividad para apreciar rápidamente dónde termina el flujo de actividad.

- Si tenemos actividades que no tienen entrada pero sí salidas, o al revés, debemos considerar nuestra representación, algo estaremos haciendo mal...

- El rombo que indica una decisión no tiene la condición escrita dentro a diferencia de los diagramas de flujo. Es en las actividades que salen de él donde escribiremos la condición evaluada y el resultado obtenido para seguir por esa transición.

- Las guardas de los rombos de decisión deben contemplar todos los casos posibles. Es correcto añadir una transición con una condición [en caso contrario].

- La paralelización (*fork*) y la sincronización (*join*) no pueden ocurrir simultáneamente. Un *fork* sólo puede tener una entrada y un *join* una salida.

2.9. Diagrama de clases

La realización de un diagrama de clases está en la frontera entre el análisis y el diseño. Probablemente es el diagrama UML más conocido (con permiso de los casos de uso), y nos permite identificar la estructura de clases del sistema incluyendo las propiedades y métodos de cada clase.

ANOTACIONES

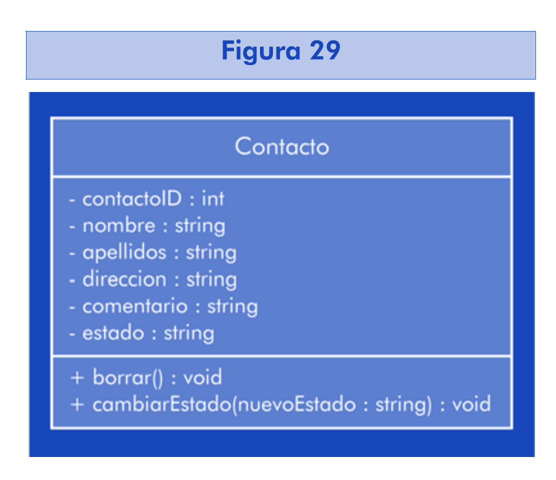
También representaremos las relaciones que existen entre las clases tales como herencia, generalización, etc., manteniendo la misma notación vista en anteriores diagramas para casos similares.

Gran parte de la popularidad de este tipo de diagrama es que numerosas herramientas de desarrollo soportan la generación de código a partir de esta representación visual, lo que facilita mucho el trabajo y evita muchos errores en las fases iniciales del proyecto. Además, algunas de estas herramientas no sólo soportan la generación inicial de código, sino que son capaces de actualizar el diagrama a partir del código fuente (ingeniería inversa) o actualizar el código a medida que vamos introduciendo cambios en el modelo aunque éste haya sido ya modificado por los desarrolladores (siempre bajo un entorno y unas condiciones especiales, claro está).

Los elementos presentes en este diagrama son únicamente las clases, y sus relaciones:

- **Clase**: se representa mediante un rectángulo dividido en tres secciones. En la parte superior deberemos indicar su nombre, a continuación sus propiedades o atributos y en la tercera sección sus métodos. Elementos auxiliares ya vistos como los estereotipos (p. ej. <<interfaz>>) también pueden aparecer junto al nombre de la clase. Los atributos y los métodos pueden incorporar información adicional como por ejemplo el tipo de acceso (público, privado, protegido), el tipo de datos de los atributos y los parámetros de los métodos, etc.

Figura 29



En la clase representada, todos los atributos son privados, y los métodos públicos.

Pueden verse indicados también los tipos de datos de unos y otros.

- **Asociación**: representa una relación genérica entre dos clases, y su notación es simplemente una línea que las une, donde podemos indicar la multiplicidad de la relación en cada extremo (uno a uno, uno a n, n a m).

- **Composición, agregación**: tal como vimos en el capítulo de introducción, si una clase está compuesta de otras, donde estas otras no pueden existir sin la primera, tendrán una relación de composición con la clase padre. Cuando simplemente una clase incluye a otra, pero la incluida tiene entidad en sí misma, hablaremos de agregación.

- **Dependencia**: cuando una clase depende de otra en el sentido de que la usa como atributo o parámetro de algún método, puede expresarse mediante una relación de dependencia.

- **Generalización**: es el equivalente a la herencia o extensión tal como hemos visto en otros diagramas.

Veamos un ejemplo:

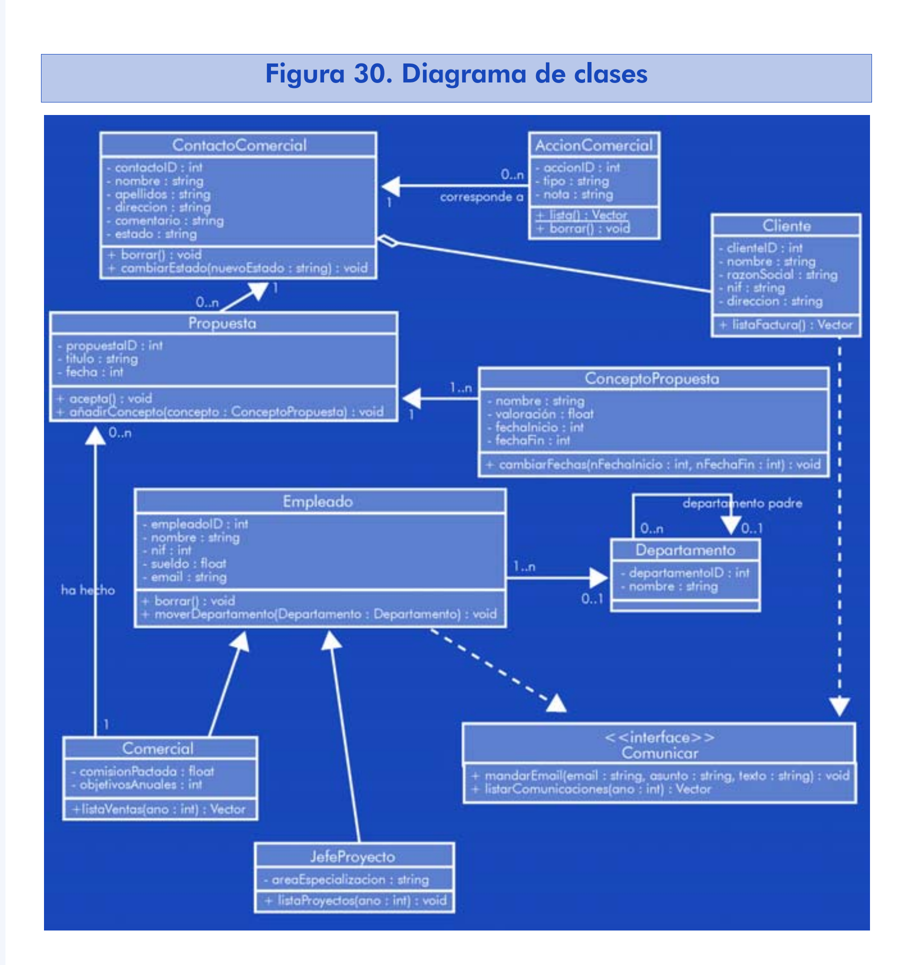


Figura 30. Diagrama de clases

Puede verse que las relaciones de generalización y agregación usan la misma notación vista en otros diagramas, lo que contribuye a la coherencia entre los modelos y facilita su comprensión y representación. En algunos casos, hemos creído necesario incluir etiquetas en las relaciones, ya que clarifican el diagrama. En otros casos (como veremos en el capítulo de generación de código) puede ser interesante representar también el rol que juega cada clase en la relación.

La relación de implementación de la interfaz, la hemos expresado con la notación de dependencia.

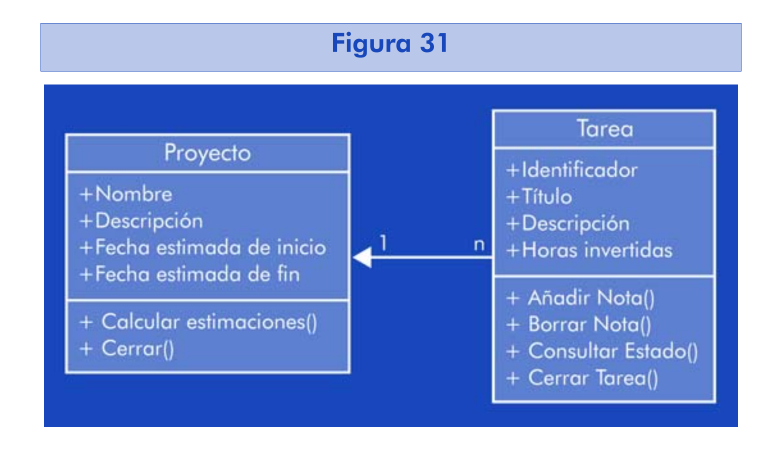
El diagrama expresa también perfectamente las clases que se tienen a ellas mismas como uno de sus atributos, como es el caso de "Departamento".

ANOTACIONES

El modelado de clases no presenta ninguna dificultad en cuanto a la representación en sí misma, la dificultad está en identificar las relaciones existentes entre las clases que componen el sistema. Para ello es imprescindible tener muy claros los conceptos vistos en el apartado correspondiente a la introducción a la orientación a objetos y consultar la numerosa bibliografía existente.

A más alto nivel, es posible representar los diagramas de clases sin información relativa a su implementación, de una forma similar a la siguiente:

Figura 31



Esta representación puede ser muy útil en fases iniciales, y si se hace, facilita mucho la creación del diagrama de clases detallado cuando llegue el momento.

A continuación, veamos algunas de las buenas prácticas a tener en cuenta al representar diagramas de clases, aunque algunas de ellas hacen referencia también al diseño de las propias clases:

- La visibilidad de los atributos (público: +; protegido: #; privado: -) es recomendable usarla sólo en fase de diseño. Se trata de un aspecto importante en el diseño del objeto y no debería obviarse, pero en diagramas conceptuales no es necesario, ya que quizá en fases posteriores habrá métodos que se conviertan en atributos y al contrario. El mismo argumento puede usarse para los tipos de datos.

- Los nombres de los métodos y los atributos deberían reflejar las convenciones en cuanto a denominación del lenguaje de programación en que vayamos a implementar el sistema. Es importante

ANOTACIONES

no sólo por coherencia, sino por aprovechar las ventajas que nos ofrecerán los generadores de código.

- Cuando una asociación tiene métodos o atributos, deberemos modelarla como una clase que se asocia con las dos anteriores.

- No es necesario incluir los métodos de acceso y modificación de los atributos (típicamente getXXX y setXXX). Pueden darse por supuestos en el modelo y la mayoría de generadores de código los generan de forma automática o bajo demanda.

- Si debemos dejar alguna lista de parámetros incompleta, indicarlo mediante una elipsis (...).

- Los atributos y operaciones estáticas (representadas con el nombre subrayado en el diagrama) deberían aparecer antes que las de instancia (el resto). Siguiendo el mismo criterio, es conveniente ordenar los métodos y atributos según su visibilidad descendiente: públicos → protegidos → privados.

- No conviene abusar de los estereotipos. Algunas herramientas incluyen estereotipos del tipo <<constructor>> o <<getter>> en los métodos correspondientes. Esto sólo contribuye a añadir complejidad al diagrama sin proporcionar información valiosa.

- Si nuestro modelo incluye múltiples interfaces, puede ser útil incluirlos en otro diagrama y utilizar la notación para la clase que implemente la interfaz del diagrama de componentes en el diagrama de clases (el círculo cerrado). Evidentemente, se desaconseja repetir los atributos y métodos de la interfaz que implementa una clase.

- Es aconsejable indicar siempre la multiplicidad de una relación. Nos será útil para entender mejor el tipo de relación y ayudará también al generador de código. Conviene ser escrupuloso en su especificación, y concretar si la relación es 0..1, 1, 0..n, 1..n, o n..m.

- No es necesario modelar todas las dependencias. Esto da como resultado diagramas completamente saturados de asociaciones que no aportan información. Es más conveniente modelar sólo las que aportarán algo al modelo.

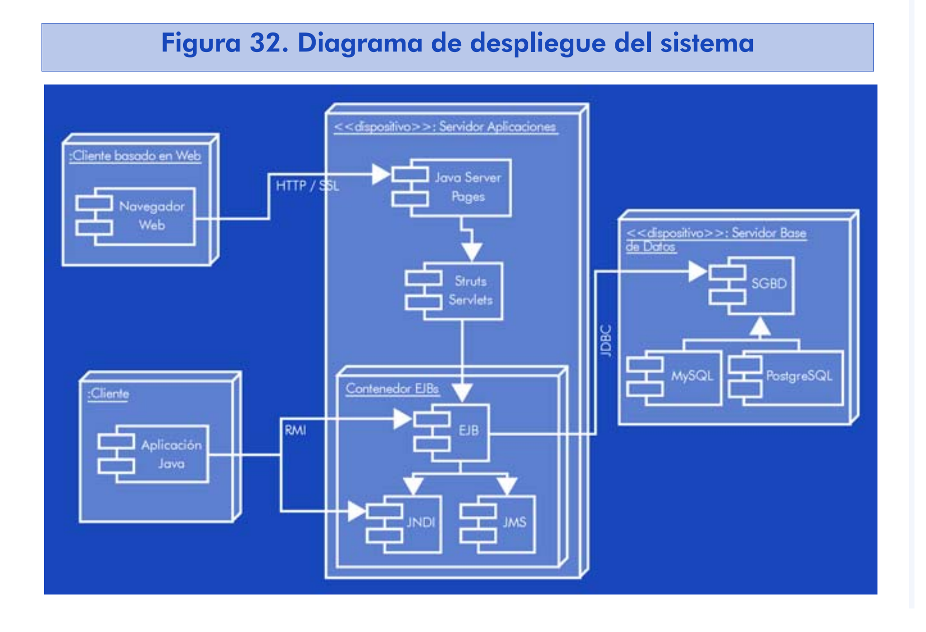
- Si existen varias relaciones entre dos clases, puede ser interesante indicar el rol que ocupan ellas en cada una. El rol se indica junto a la multiplicidad mediante el sujeto de la oración que contendría el verbo de la asociación (p. ej. "nueva propuesta", "antiguo cliente", etc.).

2.10. Diagrama de despliegue

El diagrama UML de despliegue representa una vista estática de la configuración en tiempo de ejecución de los nodos que intervienen en el proceso y de los componentes que se ejecutan en esos nodos. Puede mostrar el *hardware* del sistema y los paquetes software que hay instalados en él, el *middleware* que los conecta, etc.

También son útiles para explorar la arquitectura de sistemas embebidos, mostrando los componentes *hardware* y *software*, así como sus protocolos o formas de comunicarse.

Figura 32. Diagrama de despliegue del sistema



En el diagrama de despliegue anterior, las cajas tridimensionales representan los nodos del sistema, tanto software como hardware. Podemos clarificar de qué se trata exactamente cada uno mediante estereotipos, siempre que éstos ayuden a entender el diagrama.

Las conexiones entre nodos del sistema pueden etiquetarse con el protocolo de comunicación en que se implementan. Los nodos pueden contener subnodos, o componentes software. Los componentes software usan la misma notación que en los diagramas de componentes, por lo que podríamos añadir la notación acerca de sus interfaces si fuera necesario, aunque al nivel al que se representan los diagramas de despliegue no suele serlo.

En configuraciones complejas, los diagramas de despliegue pueden extenderse rápidamente si indicamos todos los ficheros de configuración, especificaciones de arquitecturas y versiones de cada componente implicado. Ésta es la información que sería realmente útil para los desarrolladores que deben implantar o mantener el sistema, pero no parece que UML proporcione las herramientas para representarlo de forma compacta.

Veamos el mismo ejemplo más detallado para comprobarlo.

Figura 33. Diagrama de despliegue con alguna información adicional acerca de la configuración del sistema

Vemos que la inclusión de ficheros de configuración, versiones, especificaciones de despliegue, etc. empieza a complicar el diagrama cuando quizá una simple hoja de requisitos y unas instrucciones de despliegue textuales junto con el primer diagrama serían más útiles.

Así pues, cuando trabajemos con diagramas de componentes es recomendable seguir estas prácticas:

- Indicar sólo los componentes software clave para el proyecto.

- Es importante denominar los nodos con nombres descriptivos que sean útiles para los desarrolladores.

- Intentar no abusar de los estereotipos, y mantener una coherencia en su nomenclatura con el resto de diagramas.

- Las comunicaciones también pueden incorporar estereotipos, indicando por ejemplo si la comunicación es síncrona o asíncrona, si se trata de un servicio web, etc.

No es necesario modelar todas las dependencias de los componentes dentro de un nodo. Se entiende que por ejemplo los ficheros de configuración o especificaciones de entorno de ejecución van a ser leídos por el nodo u otro componente del mismo. El desarrollador lo sabrá y en la mayoría de los casos no será necesario representar asociaciones de estos componentes con el resto.

2.11. Diagrama de estados

El propósito de los diagramas de estados es documentar las diferentes modalidades (los estados) por las que una clase puede pasar y los eventos que provocan estos cambios de estado. A diferencia de los diagramas de actividades o de secuencia que muestran las transiciones e interacciones entre clases, habitualmente el diagrama de estados muestra las transiciones dentro de una misma clase.

Normalmente lo usaremos en combinación con los casos de uso, para tener acotados los casos que provocarán cambios de estado en un objeto. No todas las clases van a necesitar un diagrama de este tipo, y normalmente van a usarse como complemento de los diagramas de actividades y de los de colaboración.

ANOTACIONES

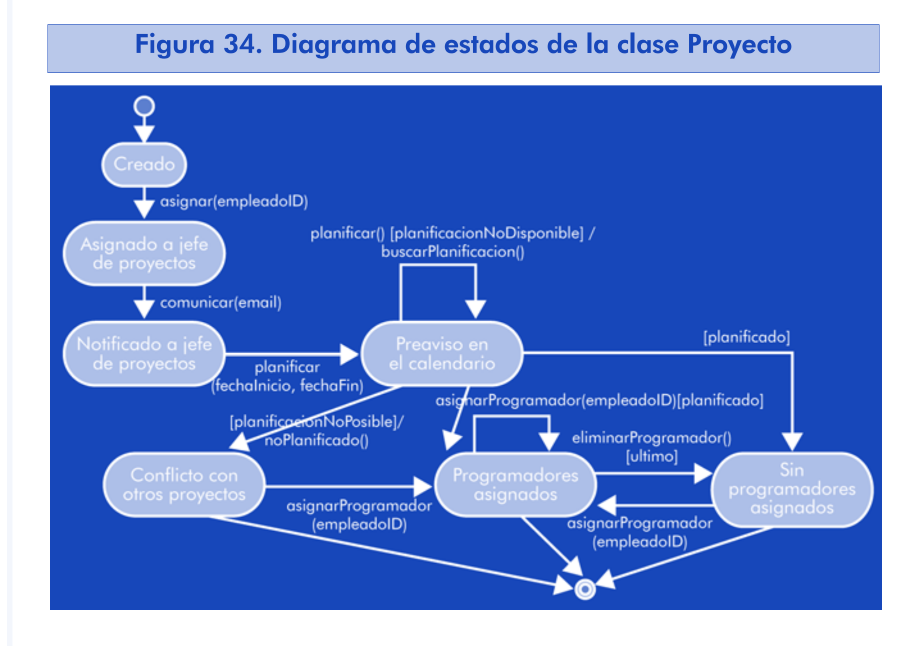
En cuanto a la notación, comparten muchos elementos con otros diagramas que representan el comportamiento del modelo, como los diagramas de actividad y colaboración ya mencionados.

- **Estado**: representa el estado de un objeto en un instante de tiempo. Tendremos tantos símbolos de estado en el diagrama como estados diferentes para cada objeto haya que modelar. Su apariencia es similar a la representación de una clase, pero con las esquinas redondeadas.

- **Estados inicial y final**: son pseudoestados que mostrarán el punto de inicio y final del flujo de actividad. Su condición de pseudoestado viene dada por el hecho de que no tiene variables ni acciones definidas.

- **Transiciones**: una flecha indicará la transición entre estados. En ella describiremos el evento que ha disparado la transición, y la acción que provoca el cambio. Existen transiciones en las que no existe un evento que las provoque (por ejemplo, ha finalizado una actividad que estaba realizando).

Figura 34. Diagrama de estados de la clase Proyecto



El ejemplo muestra las transiciones que ocurren en el objeto "Proyecto" durante su creación. En este caso, la transición puede mostrar únicamente la acción que se realiza, el evento que ocurre o la condición

para que se realice la acción en concreto. En su notación más completa, se representa de la forma siguiente:

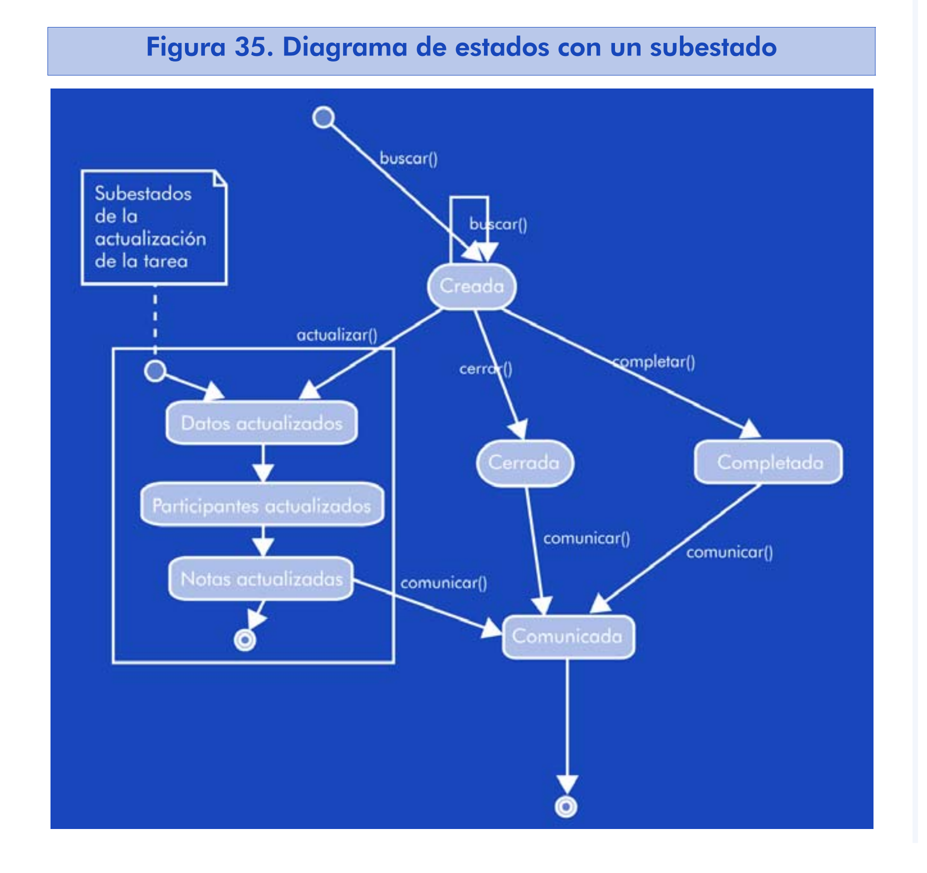
```
evento() [condición] acción()
```

Vemos que una acción puede ir iterando sobre el mismo estado según una condición. En el ejemplo podemos ir asignando programadores al proyecto, mientras el estado del objeto sigue reflejando que tiene programadores asignados, en el momento en que eliminamos el último programador, éste pasa al estado "Sin programadores asignados" del que puede salir mediante la acción `asignarProgramador()`.

Al examinar el diagrama, nos damos cuenta rápidamente de lo que comentábamos al inicio: el diagrama de estados es una representación con mucho detalle, y sólo será necesaria en casos donde los diagramas de casos en los que interviene un objeto o los diagramas de secuencia en los que interviene no proporcionen toda la información necesaria.

Al igual que en otros diagramas, puede ser interesante encapsular los estados en zonas que clarifiquen la acción global que estamos realizando o en subestados de la misma. Veamos otro ejemplo:

Figura 35. Diagrama de estados con un subestado



ANOTACIONES

En este caso se aprecian algunas incoherencias en la representación, producto de la herramienta que hemos utilizado. Teóricamente, la transición hacia el subestado debería hacerse hacia el estado inicial de la misma, pero la herramienta impide realizar una transición hacia un estado inicial. Lo mismo ocurre con el estado final del subestado, debería haber la transición hacia el estado "Comunicada", pero la herramienta impide que haya transiciones desde un estado final.

Veamos las buenas prácticas relacionadas con los diagramas de estados:

- Ordenar de arriba abajo los estados y sus transiciones (en concreto el estado inicial y final) ayuda a la comprensión del diagrama.

- Para incrementar el valor de comunicación del diagrama, es importante prestar atención a los nombres de los estados. Se aconseja usar verbos en presente (Actualizar, Presentar) o bien en participio (Actualizado, Presentado).

- Al igual que en los diagramas de actividad, habrá que vigilar con los estados que sólo emiten transiciones pero no provienen de ninguna, y de los estados que sólo reciben transiciones pero no expresan un cambio de estado del objeto.

- Si es posible, suele ser conveniente nombrar las acciones con los métodos que van a hacer cambiar el estado del objeto. De esta forma se incrementa la coherencia del diagrama con el resto.

- Si en un estado todas sus transiciones de salida o de entrada corresponden a la misma acción, y sólo varían en su condición, es correcto situar el nombre de la acción dentro del estado y sólo indicar la condición en cada transición.

- De forma similar a los diagramas de actividad, deberemos repasar las condiciones de las transiciones para asegurarnos de que no hay un caso en que pueda haber dos transiciones posibles.

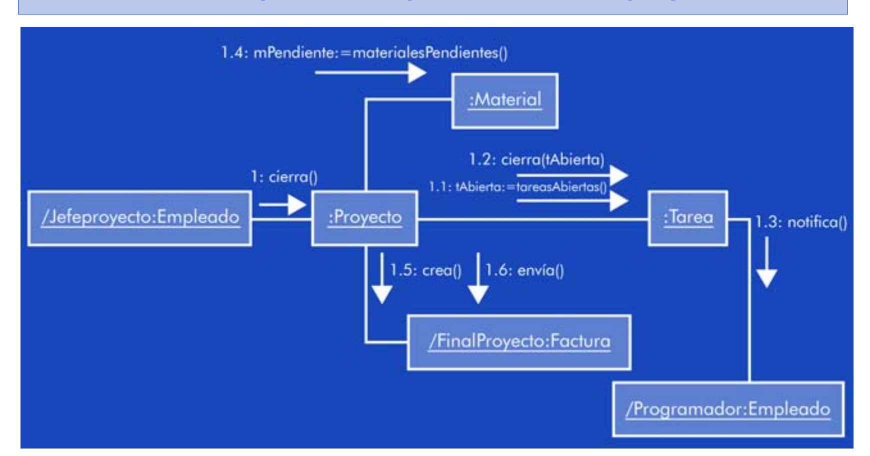
ANOTACIONES

2.12. Diagrama de colaboración

Al igual que otros diagramas UML basados en el comportamiento del sistema, los diagramas de colaboración modelan las interacciones entre objetos. Muchos autores lo consideran una variación de los diagramas de secuencia donde los objetos no están en filas y columnas, sino distribuidos libremente y con los mensajes numerados para seguir las posibles secuencias de mensajes.

Los elementos que intervienen son Objetos, Actores y Mensajes, en la misma notación que en anteriores diagramas.

Figura 36. Diagrama de colaboración representando los mensajes entre objetos al cerrar un proyecto



Vemos rápidamente que el nivel de información es prácticamente idéntico al de un diagrama de secuencia, con las siguientes excepciones:

- No se contemplan los retornos ni los errores. Simplemente los mensajes que se intercambian los objetos entre sí.
- La secuencia es más difícil de seguir, hay que mirar la etiqueta de los mensajes.

Por claridad, la notación de un diagrama de colaboración es más simple que en un diagrama de secuencia, y puede ser conveniente utilizarlo cuando el orden de los mensajes no es importante. Aprovechando su similitud, podemos también ampliar su notación con elementos de los diagramas de secuencia (las condiciones entre [ ], la

ANOTACIONES

llamada a casos de uso, etc.) y obtener un diagrama más completo y seguramente más entendible por personal no técnico que un diagrama de secuencia.

Un aspecto importante de los diagramas de colaboración es que muestran los roles que toma cada clase en la comunicación. En el diagrama puede apreciarse dos instancias de la clase "Empleado" que ocupan los roles "JefeProyecto" y "Programador".

Una variación de este diagrama toma el nombre a veces de diagrama de comunicación, en el que especificamos los objetos que se comunican entre sí, el rol que ocupa cada uno en la comunicación y la multiplicidad de unas respecto de las otras.

Podría decirse que este diagrama es una versión preliminar del diagrama de clases, donde reflejamos también la comunicación entre las mismas. La notación de los mensajes es particular para este tipo de diagramas y tiene la forma siguiente:

```
[numeroSecuencia:] nombreMetodo(parametros) [:valorRetorno]
```

o a veces la siguiente:

```
[numeroSecuencia:] [valorRetorno:=]nombreMetodo(parametros)
```

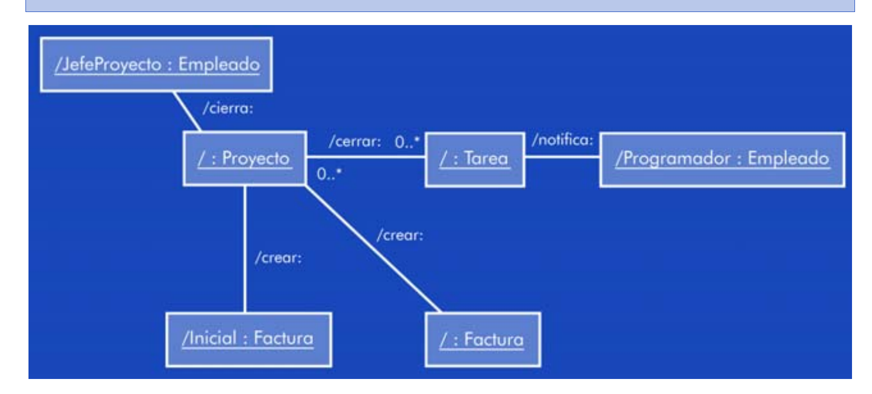
Veamos algunas buenas prácticas relacionadas con ellos:

- Los diagramas de colaboración a nivel de instancia (como el del ejemplo) son útiles para explorar posibles problemas en el diseño de las clases.

- Existe una variante a más alto nivel de este diagrama (a nivel de especificación), que suele usarse para explorar los roles que ocuparán las clases en el sistema. Esta notación no es muy habitual, ya que es casi una copia de la notación del diagrama de clases pero sin representar sus métodos y atributos.

ANOTACIONES

Figura 37. Diagrama de colaboración a nivel de especificación



- Cabe recordar que los diagramas de colaboración no modelan el flujo de proceso, únicamente las interacciones entre objetos, y esa interacción se produce a través de los mensajes que intercambian. Si deseamos modelar el proceso o los datos que intercambian, debemos utilizar un diagrama de actividades.

- Cuando sea clave la secuencia en que se producen los mensajes para comprender su interacción, es mucho más conveniente utilizar un diagrama de secuencia.

- Cuando representemos los mensajes, es importante pintar una flecha para cada mensaje, y sólo incluir los datos necesarios para comprender la comunicación. Si los parámetros o los valores de retorno no aportan nada a la comprensión del mismo, no son necesarios.

2.13. Generación de código

Hasta el momento, hemos hablado de la representación del modelo que nos proporciona UML, de su notación y de las herramientas que nos ayudan a obtener diagramas y documentación para nuestros proyectos basándonos en este estándar.

En los primeros pasos de UML, fue una empresa (Rational Corp.) la que patrocinó el proyecto, y posteriormente fueron muchas otras las que participaron y participan en el comité que revisa las aportaciones e ideas. De la misma manera que los teóricos en modelado y representación aportaron sus ideas para la notación, las empresas se preocuparon de que ésta fuera lo suficientemente completa como para que

ANOTACIONES

sus herramientas de modelado y desarrollo pudieran generar código a partir de los modelos.

La mayoría de las empresas que participan en la revisión del estándar comercializaban entornos de desarrollo que ya disponían de herramientas de generación de código y vieron el potencial de UML y las posibles mejoras en sus herramientas. Lamentablemente, ninguna de estas herramientas es de código abierto, y por lo tanto no las vamos a ver aquí, pero al ser UML un estándar abierto, muchas otras herramientas de modelado de código abierto (también excelentes) han ido incorporando esta prestación a lo largo de los años.

Las tres herramientas que hemos utilizado a lo largo del capítulo para representar los diagramas tienen, en mayor o menor medida, soporte para la generación de código a partir de los modelos.

2.13.1. Dia con Dia2Code

El software de modelado Dia disponible en http://www.gnome.org/projects/dia/ no incorpora directamente una herramienta de generación de código, pero existen varias herramientas que trabajan conjuntamente con él, ya sea para generar diagramas en su formato, como para extraer información de sus diagramas y trabajar con ella. Éste es el caso de Dia2Code.

Dia2Code está disponible en http://dia2code.sourceforge.net/ y la última versión estable es la 0.81. Es un programa ejecutable desde línea de comandos que toma los siguientes parámetros:

```
Usage: dia2code [-h|--help] [-d <dir>] [-nc] [-cl <classlist>]
       [-t (ada|c|cpp|idl|java|php|python|shp|sql)] [-v]
       [-l <license file>] <diagramfile>
  -h --help           Print this help and exit
  -t <target>         Selects the output language. <target> can be
                      one of: ada,c,cpp,idl,java,php,python,shp or sql.
                      Default is C++
  -d <dir>            Output generated files to <dir>, default is "."
  -l <license>        License file to prepend to generated files.
  -nc                 Do not overwrite files that already exist
  -cl <classlist>     Generate code only for the classes specified in
                      the comma-separated <classlist>.
                      E.g: Base,Derived.
```

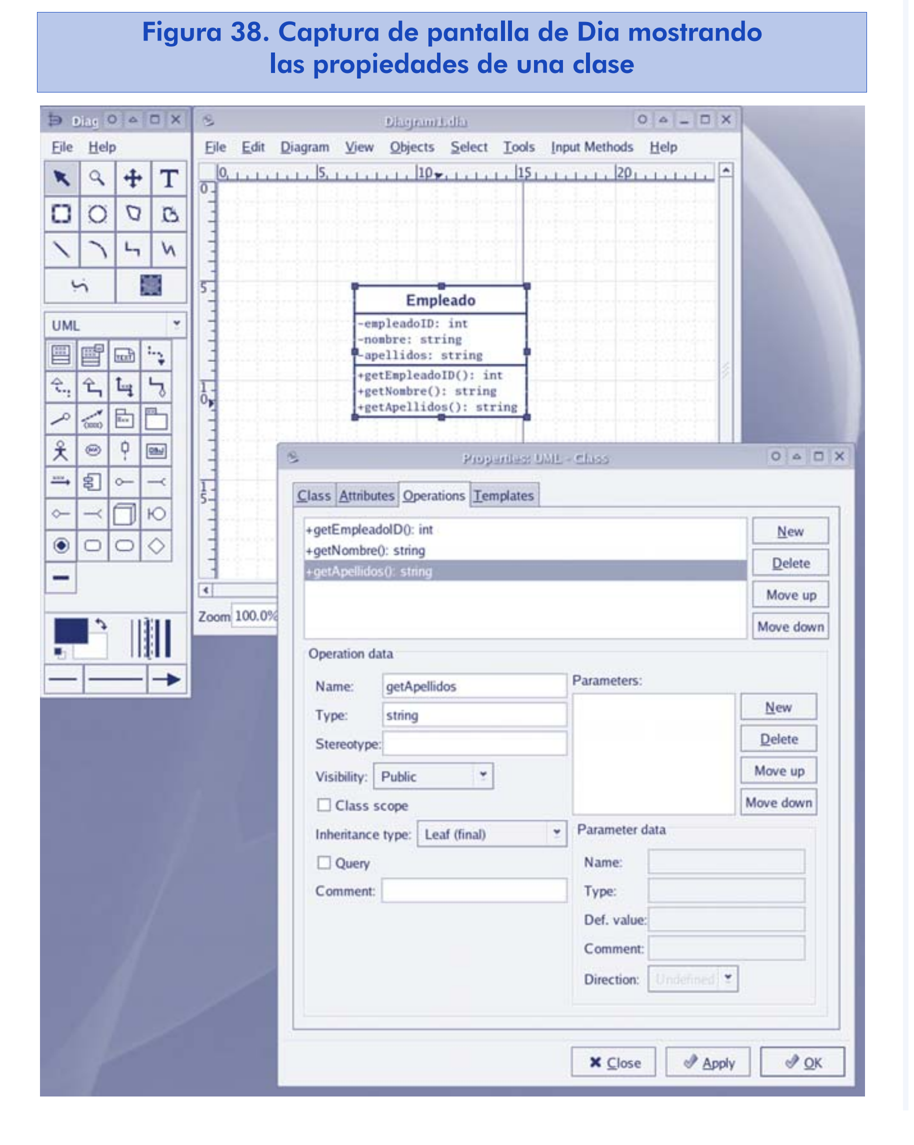
ANOTACIONES

```
-v                  Invert the class list selection. When used
                    without -cl prevents any file from being created
<diagramfile>       The Dia file that holds the diagram to be read
Note: parameters can be specified in any order.
```

A primera vista, vemos que el número de lenguajes de programación soportados es notable. Particularmente interesante es el soporte de Python y PHP, no muy habituales en estas herramientas, ni tan siquiera en las comerciales.

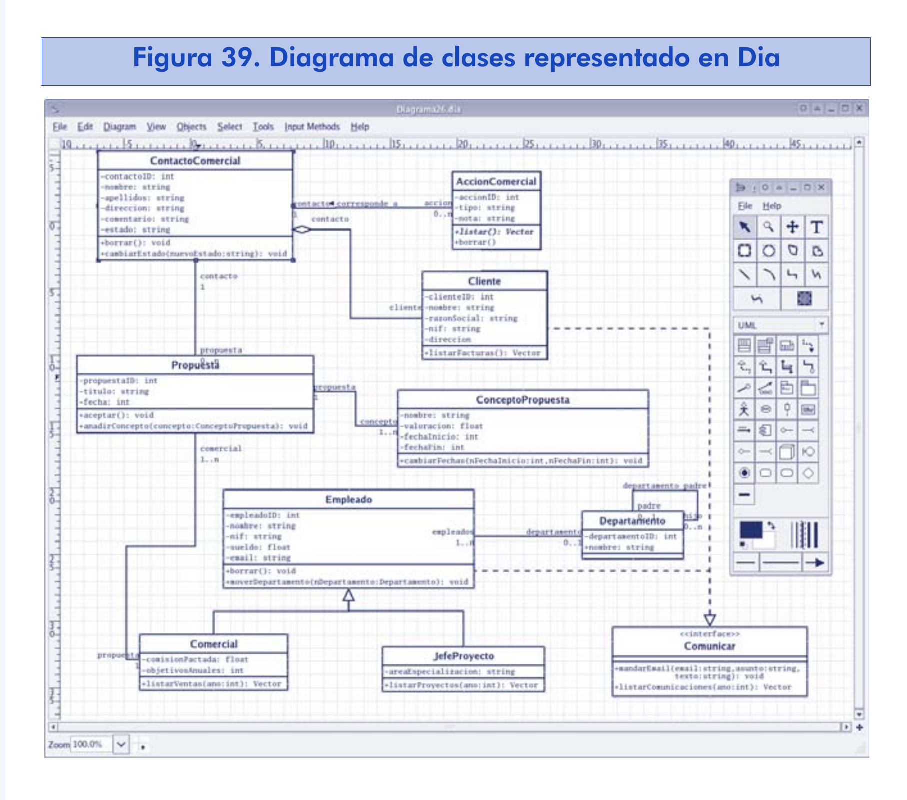
Cabe decir que para que Dia2Code interprete correctamente el diagrama UML, éste debe realizarse con las herramientas específicas UML que proporciona Dia, no pintando los mismos símbolos. Esto parece evidente pero es importante por que Dia es una herramienta de diagramas genérica y sería posible obtener el mismo resultado representando los elementos con otras formas de su biblioteca:

Figura 38. Captura de pantalla de Dia mostrando las propiedades de una clase



El diagrama sobre el que vamos a generar código es el que presentamos en el capítulo correspondiente al diagrama de clases. Para que la generación de código sea realmente útil, es conveniente proporcionar toda la información sobre las clases, métodos, atributos y cada una de las asociaciones. Así pues, conceptos como el rol que juega cada extremo de las asociaciones son importantes, ya que es presumible que el programa generador las utilice para generar el atributo correspondiente a la asociación. Veamos cómo queda representado el diagrama completo en Dia:

Figura 39. Diagrama de clases representado en Dia



Y ahora lanzamos el programa generador de código sobre el fichero del diagrama. Para este ejemplo vamos a generar código Java:

```
[marcg@lynx src]$ dia2code -t java -d /tmp/uocFiles/ Diagrama26.dia
```

Comprobamos que se han generado los ficheros para las clases. A continuación veamos los casos más interesantes y cuál ha sido el código obtenido:

```
public class Propuesta {
   /** Attributes */
   private int propuestaID;
   private string titulo;
   private int fecha;
```

```
   /** Associations */
   private Comercial propuesta;
   /**
   * Operation
   *
   */
   public void aceptar ( ){
   }
   /**
   * Operation
   *
   * @param concepto
   */
   public void anadirConcepto ( ConceptoPropuesta concepto ){
   }
}

import Empleado;

public class Comercial extends Empleado {
   /** Attributes */
   private float comisionPactada;
   private int objetivosAnuales;
   /**
   * Operation
   *
   * @param ano
   * @return Vector
   */
   public Vector listarVentas ( int ano ){
   }
}

import Comunicar;

public class Cliente implements Comunicar {
   /** Attributes */
   private int clienteID;
   private string nombre;
   private string razonSocial;
   private string nif;
   private direccion;
   /**
   * Operation
   *
   * @return Vector
   */
   public Vector listarFacturas ( ){
   }
}


import Propuesta;
import Cliente;
import AccionComercial;

public class ContactoComercial {
   /** Attributes */
```

ANOTACIONES

```
    private int contactoID;
    private string nombre;
    private string apellidos;
    private string direccion;
    private string comentario;
    private string estado;
    /** Associations */
    private Propuesta propuesta;
    private Cliente cliente;
    private AccionComercial accion;
    /**
    * Operation
    *
    */
    public void borrar ( ){
    }
    /**
    * Operation
    *
    * @param nuevoEstado
    */
    public void cambiarEstado ( string nuevoEstado ){
    }
}
```

Podemos ver cómo se han generado correctamente las importaciones entre clases, los atributos y los métodos. Destacable es también la inclusión de comentarios al estilo JavaDoc.

Por lo que respecta a las particularidades del modelo, el generador ha entendido perfectamente las relaciones de herencia (“extends”), la generalización y la implementación de la interfaz por parte de la clase “Cliente”. En todos los casos ha incluido la declaración de atributos con los nombres de roles que hemos representado en el diagrama. Por lo que respecta a la multiplicidad en las asociaciones, es lógico que la generación no incluya atributos para las asociaciones con multiplicidad mayor que uno, ya que éstas se implementaran mediante métodos (p. ej. `anadirContacto(),borrarContacto()`) y no con vectores u otras estructuras de datos estáticas (aunque otros generadores toman enfoques diferentes, como veremos).

Dia ha modelado todos los métodos de cada clase, pero no ha incluido métodos getXXX y setXXX para cada atributo. Si queremos que el generador los incluya en el código fuente, deberemos representarlos en el modelo, a pesar de añadir información irrelevante al diagrama.

ANOTACIONES

2.13.2. Umbrello

El software de modelo Umbrello disponible en http://uml.sourceforge.net o como parte del paquete "kdesdk" del entorno KDE, ha evolucionado rápidamente durante los últimos años y actualmente mantiene un alto grado de actividad en su proyecto.

A diferencia de Dia, Umbrello es una aplicación totalmente orientada al modelado UML y por ello dispone de herramientas que facilitan mucho la creación de todos los tipos de diagramas. Entre ellas, destaca la barra lateral izquierda donde aparecen todos los elementos UML que hemos creado en algún diagrama para poder aprovecharlos en los nuevos diagramas que realicemos del mismo modelo.

La generación de código es muy completa, y disponemos de un asistente que nos presenta multitud de opciones. A continuación podemos ver las disponibles para la generación de código Java:

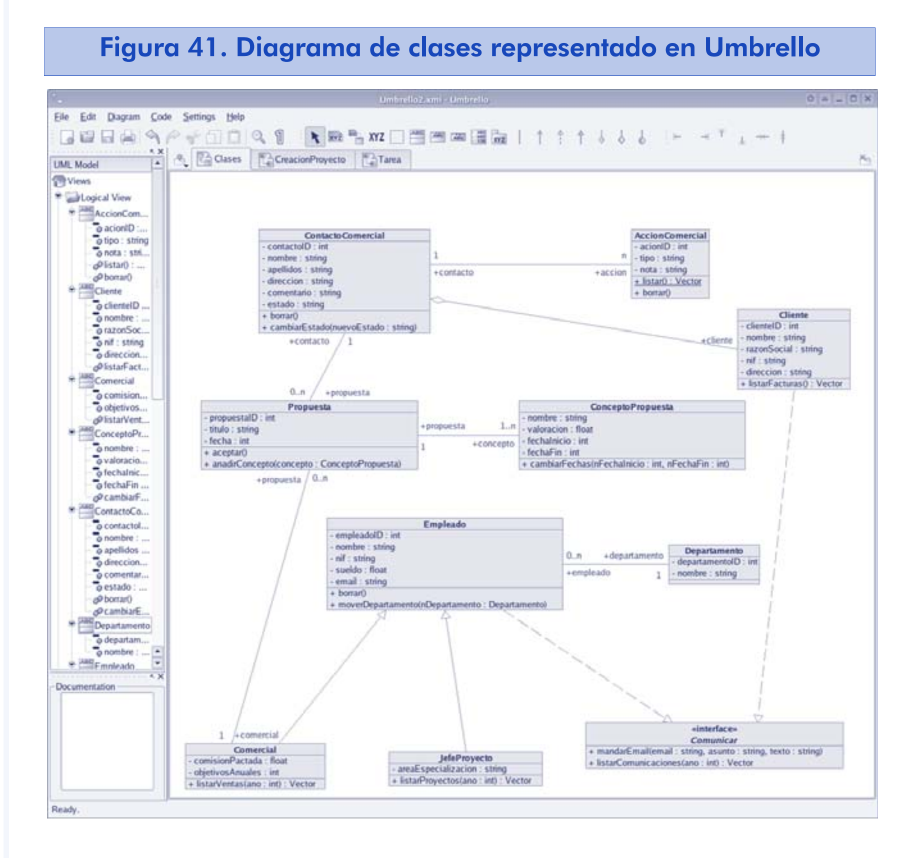
Figura 40. Cuadro de diálogo con las opciones de generación de código Java en Umbrello



ANOTACIONES

Éste es el aspecto del diagrama de clases representado en Umbrello:

Figura 41. Diagrama de clases representado en Umbrello



El código fuente de las clases generadas es muy completo. Los comentarios están en formato JavaDoc, y aunque no aprovechemos las opciones de documentación de las clases y métodos que ofrece el propio Umbrello, él ya genera mucha información.

```
import Empleado;
import Propuesta;
/**
 * Class Comercial
 *
 */
public class Comercial extends Empleado {
 // Fields
 //
 private float comisionPactada;
 //
 private int objetivosAnuales;
 //
 public List propuestaVector = new Vector( );
 // Methods
 // Constructors
 // Empty Constructor
 public Comercial ( ) { }
 // Accessor Methods
```

ANOTACIONES

```
 /**
  * Get the value of comisionPactada
  *
  * @return the value of comisionPactada
  */
 private float getComisionPactada ( ) {
  return comisionPactada;
 }
 /**
  * Set the value of comisionPactada
  *
  *
  */
 private void setComisionPactada ( float value ) {
  comisionPactada = value;
 }
 /**
  * Get the value of objetivosAnuales
  *
  * @return the value of objetivosAnuales
  */
 private int getObjetivosAnuales ( ) {
  return objetivosAnuales;
 }
 /**
  * Set the value of objetivosAnuales
  *
  *
  */
 private void setObjetivosAnuales ( int value ) {
  objetivosAnuales = value;
 }
 /**
  * Add an object of type Propuesta to the List propuestaVector
  *
  * @return void
  */
public void addPropuesta ( Propuesta value ) {
   propuestaVector.add(value);
}
/**
  * Remove an object of type Propuesta from the List propuestaVector
  *
  */
public void removePropuesta ( Propuesta value ) {
   propuestaVector.remove(value);
}
/**
 * Get the list of propuestaVector
 *
 * @return List of propuestaVector
 */
 public List getPropuestaList ( ) {
  return (List) propuestaVector;
 }
 // Operations
```

ANOTACIONES

```
  /**
   *
   * @param ano
   * @return Vector
   */
  public Vector listarVentas ( int ano) {

  }
}


import ContactoComercial;
import Comunicar;
/**
 * Class Cliente
 *
 */
public class Cliente implements Comunicar {
  // Fields
  //
  private int clienteID;
  //
  private String nombre;
  //
  private String razonSocial;
  //
  private String nif;
  //
  private String direccion;
  //
  public ContactoComercial contacto = new ContactoComercial ( );
  // Methods
  // Constructors
  // Empty Constructor
  public Cliente ( ) { }
  // Accessor Methods
  /**
   * Get the value of clienteID
   *
   * @return the value of clienteID
   */
  private int getClienteID ( ) {
   return clienteID;
  }
  /**
   * Set the value of clienteID
   *
   *
   */
  private void setClienteID ( int value ) {
   clienteID = value;
  }
  /**
   * Get the value of nombre
   *
   * @return the value of nombre
   */
```

ANOTACIONES

```
private String getNombre ( ) {
 return nombre;
}
/**
 * Set the value of nombre
 *
 *
 */
private void setNombre ( String value ) {
 nombre = value;
}
/**
 * Get the value of razonSocial
 *
 * @return the value of razonSocial
 */
private String getRazonSocial ( ) {
 return razonSocial;
}
/**
 * Set the value of razonSocial
 *
 *
 */
private void setRazonSocial ( String value ) {
 razonSocial = value;
}
/**
 * Get the value of nif
 *
 * @return the value of nif
 */
private String getNif ( ) {
 return nif;
}
/**
 * Set the value of nif
 *
 *
 */
private void setNif ( String value ) {
 nif = value;
}
/**
 * Get the value of direccion
 *
 * @return the value of direccion
 */
private String getDireccion ( ) {
 return direccion;
}
/**
 * Set the value of direccion
 *
 *
 */
```

ANOTACIONES

```
 private void setDireccion ( String value ) {
  direccion = value;
 }
 /**
  * Get the value of contacto
  *
  * @return the value of contacto
  */
 public ContactoComercial getContacto ( ) {
  return contacto;
 }
 /**
  * Set the value of contacto
  *
  *
  */
public void setContacto ( ContactoComercial value ) {
  contacto = value;
}
// Operations
/**
 *
 * @return Vector
 */
public Vector listarFacturas ( ) {

 }
}
```

Hemos marcado las opciones para generar los métodos de acceso a los atributos, y así lo ha hecho, proporcionando además comentarios que aunque triviales, sientan la base para que los completemos.

Soporta adecuadamente todas las asociaciones, herencia e implementación de interfaces. Además, ha generado vectores o variables simples en las asociaciones según la multiplicidad de las mismas, y métodos para añadir, borrar y obtener la lista de la clase asociada.

Además Umbrello incorpora la posibilidad de importar las clases a partir de sus ficheros fuente. Esta opción nos permite disponer del modelo de las clases junto con sus atributos y métodos, pero no pinta el diagrama de clases correspondiente. Para ello, simplemente deberemos arrastrar las clases desde la lista situada en la parte izquierda hasta el diagrama y crear las asociaciones que Umbrello no haya detectado.

ANOTACIONES

En resumen, podemos decir que estamos ante una muy buena herramienta, que cumple con los objetivos de cualquier proyecto de desarrollo que busque un software de modelado con generación de código de calidad. Además, su facilidad de uso y la gran cantidad de lenguajes de programación soportados (cabe destacar el soporte diferenciado de PHP4 y PHP5, Perl y XMLSchema) nos permite disponer de buenos resultados en muy poco tiempo.

2.13.3. ArgoUML

El entorno de modelado ArgoUML disponible en http://argouml.tigris.org es quizá el más completo en términos de cumplimiento con el estándar. Es un proyecto que empezó en privado en 1995 como una herramienta CASE y en 1999 fue evolucionando hacia un proyecto de código abierto integrando el modelado UML y prestaciones para el desarrollo rápido de aplicaciones.

Esta orientación hacia el desarrollo es notoria no sólo en su interfaz, donde el diagrama juega un papel casi secundario, también en el soporte estricto de la notación y en prestaciones como las críticas automáticas al diseño o consejos que vamos recibiendo a medida que nuestro modelo va evolucionando.

La generación de código no ofrece tantas opciones como en los programas anteriores (sólo genera código Java y la propia notación UML 1.3), no soporta la inclusión de ficheros de licencia o cabeceras, etc. El enfoque de ArgoUML es totalmente hacia UML como notación y el diseño de aplicaciones orientadas a objeto.

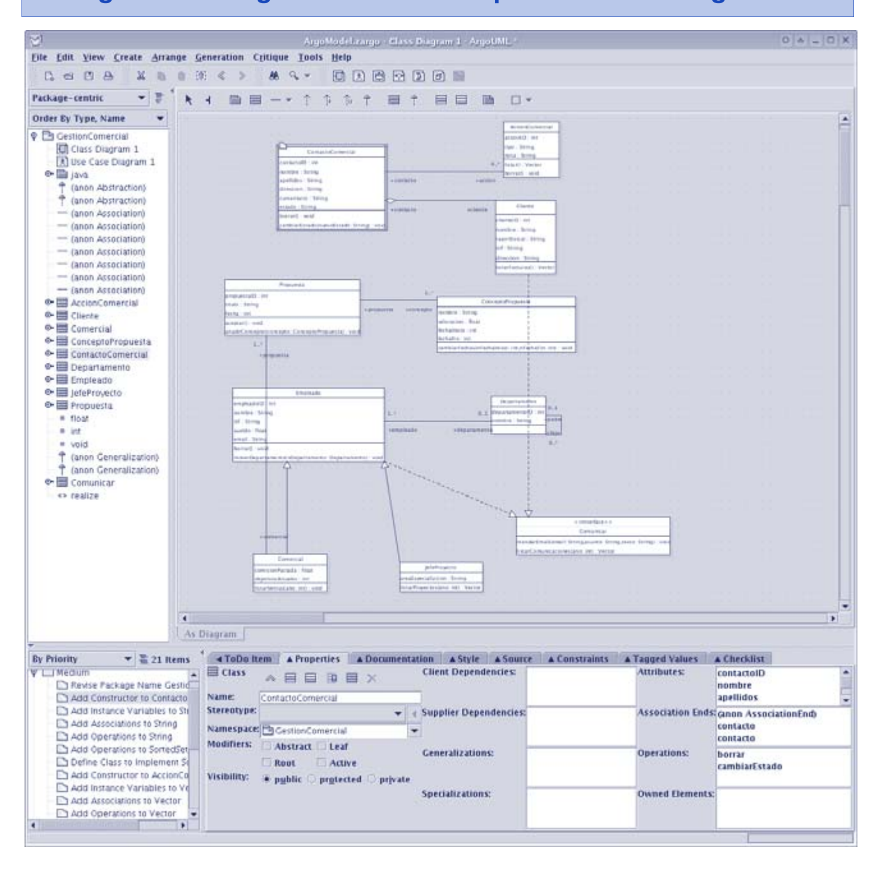
Cabe destacar especialmente la prestación de “Críticas al diseño”, donde para cada clase, atributo, método y en general cualquier elemento del modelo, ArgoUML nos presenta una *checklist* que podemos repasar y validar para mejorar nuestro diseño. También nos ofrece críticas en general sobre el diagrama en aspectos como la notación (el uso de mayúsculas y minúsculas en los nombres de clases, atributos y métodos), el uso de patrones de diseño, la inclusión de constructores, etc.

ANOTACIONES

ANOTACIONES

Éste es el aspecto de ArgoUML mostrando nuestro diagrama de clases:

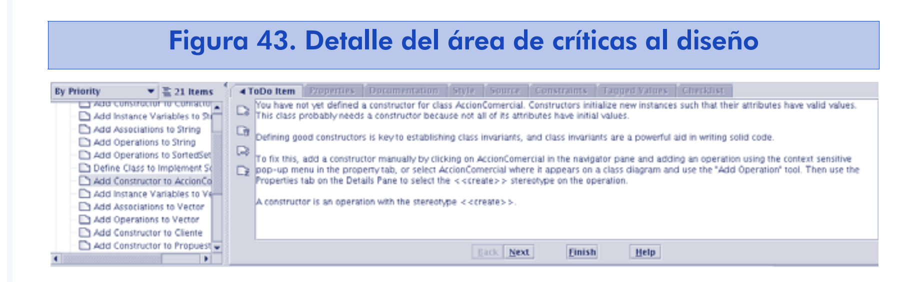
Figura 42. Diagrama de clases representado en ArgoUML



Podemos ver que en la parte izquierda del entorno disponemos del árbol de elementos UML que incluye nuestro proyecto. En la parte inferior es donde se muestran las propiedades del elemento seleccionado (en este caso una clase), y en la parte inferior izquierda tenemos los consejos de diseño.

Veamos este último aspecto con un poco más de detalle:

Figura 43. Detalle del área de críticas al diseño



Vemos que la información que ofrece para un elemento en concreto (crear el constructor de una clase) es muy extensa e incluye consejos

tanto para crear el constructor con el programa, como sobre la importancia del mismo y de las implicaciones que tiene en UML.

Como hemos comentado, la generación de código no ofrece muchas opciones, y se limita a crear simplemente los métodos representados, así pues, a diferencia de los programas evaluados anteriormente, no generará código ni para el constructor ni para los métodos de acceso a los atributos ni para acceder a las variables representadas para las asociaciones.

```
import java.util.Vector;

public class Comercial extends Empleado {
   /* {src_lang=Java}*/

   private float comisionPactada;
   /* {transient=false, volatile=false}*/

   private int objetivosAnuales;
   /* {transient=false, volatile=false}*/

     /**
     *
     * @element-type Propuesta
     */
   public Vector propuesta;

   public void listarVentas(int ano) {
   }
}


import java.lang.String;
import java.util.Vector;

public class Cliente implements Comunicar {
   /* {src_lang=Java}*/


   private int clienteID;
   /* {transient=false, volatile=false}*/

   private String nombre;
   /* {transient=false, volatile=false}*/

   private String razonSocial;
   /* {transient=false, volatile=false}*/

   private String nif;
   /* {transient=false, volatile=false}*/

   private String direccion;
   /* {transient=false, volatile=false}*/

     public ContactoComercial myContactoComercial;
```

ANOTACIONES

```
    public ContactoComercial contacto;

   public Vector listarFacturas() {
   return null;
   }
}

import java.lang.String;
import java.util.Vector;

public class ContactoComercial {
   /* {src_lang=Java}*/


   private int contactoID;
   /* {transient=false, volatile=false}*/

   private String nombre;
   /* {transient=false, volatile=false}*/

   private String apellidos;
   /* {transient=false, volatile=false}*/

   private String direccion;
   /* {transient=false, volatile=false}*/

   private String comentario;
   /* {transient=false, volatile=false}*/

   private String estado;
   /* {transient=false, volatile=false}*/

    public Cliente cliente;
    /**
    *
    * @element-type AccionComercial
    */
   public Vector accion;

   public void borrar() {
   }

   public void cambiarEstado(String nuevoEstado) {
   }
}
```

Vemos que ha generado código para las asociaciones y ha respetado correctamente la multiplicidad, creando un Vector para la multiplicidad múltiple, y una instancia para el resto.

Debido a la calidad y a la popularidad de este software en su área de especialización (el modelado de diseño), han surgido versiones comerciales mejoradas por otros fabricantes de software, entre las que cabe destacar PoseidonUML (http://www.gentleware.com/products/)

ANOTACIONES

que implementa una mucho mejor generación de código e integración con entornos de desarrollo como Eclipse.

ArgoUML también incorpora funcionalidad para la importación de ficheros fuente (sólo en Java) al modelo, de modo que podamos aprovechar esas clases en los diferentes diagramas que soporta.

2.14. Resumen

El análisis y diseño de software juega un papel crucial en cualquier desarrollo, pero es en la programación orientada a objeto donde las actividades relacionadas con esta fase de un proyecto han alcanzado sus cotas más altas de sofisticación. Es, además, un área donde continuamente se producen avances y nuevas propuestas.

En este capítulo hemos empezado dando al estudiante una visión general del paradigma de programación orientada a objeto, donde hemos repasado desde los conceptos más básicos hasta llegar a todos los tipos de relaciones entre objetos que soportan la mayoría de lenguajes de programación.

Esta introducción ha sido necesaria para poder centrarnos a continuación en el lenguaje de modelado UML y lo que éste puede aportar a lo largo de todas las fases del ciclo de vida de un proyecto. Cada uno de los diagramas se ha estudiado en detalle a partir de un caso práctico.

Aunque la mayoría de los diagramas de UML nos permiten mejorar la comunicación con el cliente en las fases iniciales, así como documentar y explorar funcionalidades y aspectos concretos de las clases de nuestro sistema durante su análisis y diseño, también hemos visto cómo UML puede ayudarnos en las fases de desarrollo e implantación.

Hemos dedicado el último apartado a la generación de código mediante las tres mismas herramientas de código abierto que hemos utilizado en el material para generar los diagramas, demostrando así que UML en particular y el modelado en general es una muy buena práctica en cualquier proyecto, no sólo por la disciplina que nos

ANOTACIONES

impone en su diseño sino también por el ahorro de tiempo en el desarrollo que puede aportar.

Por todo ello, creemos que con la lectura de este capítulo el estudiante habrá logrado los objetivos planteados al inicio del mismo.

Este material no pretende ser la referencia última de UML para el estudiante, sino una visión global de este estándar de modelado, una motivación para explorar las referencias que proporcionamos y un incentivo para poner en práctica los conocimientos y buenas prácticas que hemos indicado en los próximos proyectos en los que participe.

2.15. Otras fuentes de referencia e información

ArgoUML. http://argouml.tigris.org/

Birtwistle, G.M. (Graham M.) 1973. *SIMULA begin*. Philadelphia, Auerbach

Dia. http://www.gnome.org/projects/dia/

Eckel, B. (2003). *Thinking in Java* (3.ª ed.). Upper Saddle River: Prentice Hall. http://www.mindview.net/Books/TIJ/

Gamma, E.; Helm; R. y otros (1995). *Design Patterns*. Reading (Mass.): Addison Wesley.

Java Technology. http://java.sun.com

Microsoft.NET. http://www.microsoft.com/net/

Object Management Group. http://www.omg.org/

Rumbaugh, J. y otros (2004). *Object-Oriented Modeling and Design with UML* (2nd Edition). Englewood Cliff (N.J.): Prentice Hall.

Smalltalk. http://www.smalltalk.org

ANOTACIONES

Stroustrup, B. (2000). *The C++ Programming Language* (3º edition). Reading (Mass.): Addison Wesley.

The Open Source Java Directory. http://www.onjava.com/pub/q/java_os_directory

Umbrello. http://uml.sourceforge.net

UML1.5 The Current Official Version. http://www.uml.org/#UML1.5

Unified Modeling Languag. http://www.uml.org/

Wikipedia. http://www.wikipedia.org/

ANOTACIONES

3. Control de calidad y pruebas

3.1. Introducción

La calidad del software es una preocupación a la que se dedican muchos esfuerzos. Sin embargo, el software casi nunca es perfecto. Todo proyecto tiene como objetivo producir el software de la mejor calidad posible, que cumpla, y si puede ser supere, las expectativas de sus usuarios. Existe abundante literatura sobre los procesos de calidad de software y actualmente se realiza una considerable investigación académica en este campo dentro de la ingeniería del software.

En este capítulo intentaremos dar una visión práctica de los controles de calidad y de pruebas en el caso particular, tanto de prácticas como de aplicaciones, del software libre, y veremos cuáles son los principales principios, técnicas y aplicaciones que se utilizan para garantizar la calidad de los productos libres.

3.2. Objetivos

Los materiales didácticos asociados a este módulo permitirán al estudiante obtener los siguientes conocimientos:

- Familiarizarse con la terminología relacionada con el control de calidad y pruebas que es de uso común en la ingeniería del software.
- Conocer las principales técnicas de comprobación manual de software usadas en la ingeniería del software y en entornos libres.
- Conocer las principales técnicas de comprobación automática de software usadas en ingeniería del software y el software libre que se utiliza en ellas.

ANOTACIONES

- Conocer los sistemas de comprobación de unidades de software y entender su funcionamiento.

- Conocer los sistemas de gestión de errores, la anatomía de un error, su ciclo de vida, y familiarizarse con el sistema de gestión de errores libre Bugzilla.

3.3. Control de calidad y pruebas

Lectura recomendada

M.A. Cusumano. *Preliminary Data from Global Software Process Survey*.

Cualquier usuario o desarrollador sabe, por experiencia propia, que los programas no son perfectos. Los programas tienen errores. Cuanto mejor sea el proceso de ingeniería del software y el proceso de control de calidad y pruebas que se utilice, menos errores tendrá. El software tiene una media de 0,150 errores por cada 1.000 líneas de código. Si tenemos en cuenta que un producto como OpenOffice.org 1.0 tiene aproximadamente 7 millones de líneas de código, la aritmética es sencilla.

Hoy en día cualquier proyecto de software integra dentro de su ciclo de desarrollo procesos de control de calidad y pruebas. No existe una única práctica unánimemente aceptada, ni el mundo académico ni empresarial para llevar a cabo estos procesos. Tampoco existe ningún método que se considere mejor que los otros. Cada proyecto de software utiliza la combinación de métodos que mejor se adapta a sus necesidades.

Los proyectos que no integran un control de calidad como parte de su ciclo de desarrollo a la larga tienen un coste más alto. Esto es debido a que cuanto más avanzado está el ciclo de desarrollo de un programa, más costoso resulta solucionar sus errores. Se requieren entonces un mayor número de horas de trabajo dedicadas a solucionar dichos errores y además sus repercusiones negativas serán más graves. Por esto, siempre que iniciemos el desarrollo de un nuevo proyecto deberíamos incluir un plan de gestión de calidad.

Para poder llevar a cabo un control de calidad, es imprescindible haber definido los requisitos del sistema de software que vamos a implementar, ya que son necesarios para disponer de una especificación del

ANOTACIONES

propósito del software. Los procesos de calidad verifican que el software cumple con los requisitos que definimos originalmente.

3.3.1. Términos comunes

Las técnicas y sistemas de control de calidad y pruebas tienen una terminología muy específica para referirse a conceptos singulares de este tipo de sistemas. A continuación veremos los conceptos y términos más habituales:

- **Error** (*bug*). Fallo en la codificación o diseño de un sistema que causa que el programa no funcione correctamente o falle.

- **Alcance del código** (*code coverage*). Proceso para determinar qué partes de un programa nunca son utilizadas o que nunca son probadas como parte del proceso de pruebas.

- **Prueba de estrés** (*stress testing*). Conjunto de pruebas que tienen como objetivo medir el rendimiento de un sistema bajo cargas elevadas de trabajo. Por ejemplo, si una determinada aplicación web es capaz de satisfacer con unos estándares de servicio aceptables un número concreto de usuarios simultáneos.

- **Prueba de regresión** (*regression testing*). Conjunto de pruebas que tienen como objetivo comprobar que la funcionalidad de la aplicación no ha sido dañada por cambios recientes que hemos realizado. Por ejemplo, si hemos añadido una nueva funcionalidad a un sistema o corregido un error, comprobar que estas modificaciones no han dañado al resto de funcionalidad existente del sistema.

- **Prueba de usabilidad** (*usability testing*). Conjunto de pruebas que tienen como objetivo medir la usabilidad de un sistema por parte de sus usuarios. En general, se centra en determinar si los usuarios de un sistema son capaces de conseguir sus objetivos con el sistema que es objeto de la prueba.

- **Testeador** (*tester*). Persona encargada de realizar un proceso de pruebas, bien manuales o automáticas, de un sistema y reportar sus posibles errores.

ANOTACIONES

3.3.2. Principios de la comprobación de software

Cualquiera que sea la técnica o conjunto de técnicas que utilicemos para asegurar la calidad de nuestro software, existen un conjunto de principios que debemos tener siempre presentes. A continuación enumeramos los principales:

- **Es imperativo disponer de unos requisitos que detallen el sistema**. Los procesos de calidad se basan en verificar que el software cumple con los requisitos del sistema. Sin unos requisitos que describan el sistema de forma clara y detallada, es imposible crear un proceso de calidad con unas mínimas garantías.

- **Los procesos de calidad deben ser integrados en las primeras fases del proyecto**. Los procesos de control de calidad del sistema deben ser parte integral del mismo desde sus inicios. Realizar los procesos de pruebas cuando el proyecto se encuentra en una fase avanzada de su desarrollo o cerca de su fin es una mala práctica en ingeniería del software. Por ejemplo, en el ciclo final de desarrollo es muy costoso corregir errores de diseño. Cuanto antes detectemos un error en el ciclo, más económico será corregirlo.

- **Quien desarrolle un sistema no debe ser quien prueba su funcionalidad**. La persona o grupo de personas que realizan el desarrollo de un sistema no deben ser en ningún caso las mismas que son responsables de realizar el control de pruebas. De la misma manera que sucede en otras disciplinas, como por ejemplo la escritura, donde los escritores no corrigen sus propios textos. Es importante recordar que a menudo se producen errores en la interpretación de la especificación del sistema y una persona que no ha estado involucrada con el desarrollo del sistema puede más fácilmente evaluar si su interpretación ha sido o no correcta.

Es importante tomar en consideración todos estos principios básicos porque el incumplimiento de alguno de ellos se traduce en la imposibilidad de poder garantizar la corrección de los sistemas de control de calidad que aplicamos.

ANOTACIONES

3.4. Técnicas manuales de comprobación de software

Las técnicas manuales de comprobación de software son un conjunto de métodos ampliamente usados en los procesos de control de pruebas. En su versión más informal consisten en un grupo de testeadores que instalan una aplicación y, sin ningún plan predeterminado, la utilizan y reportan los errores que van encontrando durante su uso para que sean solucionados.

En la versión más formal de este método se utilizan guiones de pruebas, que son pequeñas guías de acciones que un testeador debe efectuar y los resultados que debe obtener si el sistema está funcionando correctamente. Es habitual en muchos proyectos que antes de liberar una nueva versión del proyecto deba superarse con satisfacción un conjunto de estos guiones de pruebas.

La mayoría de guiones de pruebas tienen como objetivo asegurar que la funcionalidad más común del programa no se ha visto afectada por las mejoras introducidas desde la última versión y que un usuario medio no encontrará ningún error grave.

Ejemplo

Éste es un ejemplo de guión de pruebas del proyecto OpenOffice.org que ilustra su uso.

Área de la prueba: Solaris - Linux - Windows

Objetivo de la prueba

Verificar que OpenOffice.org abre un fichero .jpg correctamente

Requerimientos:

Un fichero .jpg es necesario para poder efectuar esta prueba

Acciones

Descripción:

1) Desde la barra de menús, seleccione *File* - *Open*.

ANOTACIONES

2) La caja de diálogo de abertura de ficheros debe mostrarse.

3) Introduzca el nombre del fichero .jpg y seleccione el botón *Open*.

4) Cierre el fichero gráfico.

Resultado esperado:
OpenOffice.org debe mostrar una nueva ventana mostrando el fichero gráfico.

Fuente: Ejemplo de guión de pruebas extraído del proyecto OpenOffice.org. http://qa.openoffice.org/

Como vemos, se trata de una descripción de acciones que el testeador debe seguir, junto con el resultado esperado para dichas acciones.

Aunque este sistema hoy en día se usa ampliamente en combinación con otros sistemas, confiar el control de calidad únicamente a un proceso de pruebas manuales es bastante arriesgado y no ofrece garantías sólidas de la calidad del producto que hemos producido.

3.5. Técnicas automáticas de comprobación de software

3.5.1. *White-box testing*

El *white-box testing* es un conjunto de técnicas que tienen como objetivo validar la lógica de la aplicación. Las pruebas se centran en verificar la estructura interna del sistema sin tener en cuenta los requisitos del mismo.

Existen varios métodos en este conjunto de técnicas, los más habituales son la inspección del código de la aplicación por parte de otros desarrolladores con el objetivo de encontrar errores en la lógica del programa, código que no se utiliza (*code coverage* en inglés) o la estructura interna de los datos. Existen también aplicaciones propietarias de uso extendido que permiten automatizar todo este tipo de pruebas, como PureCoverge de Rational Software.

ANOTACIONES

3.5.2. Black-box testing

El *black-box testing* se basa en comprobar la funcionalidad de los componentes de una aplicación. Determinados datos de entrada a una aplicación o componente deben producir unos resultados determinados. Este tipo de prueba está dirigida a comprobar los resultados de un componente de software no a validar como internamente ha sido estructurado. Se llama *black box* (caja negra) porque el proceso de pruebas asume que se desconoce la estructura interna del sistema, sólo se comprueba que los datos de salida producidos por una entrada determinada son correctos.

Imaginemos que tenemos un sistema orientado a objetos donde existe una clase de manipulación de cadenas de texto que permite concatenar cadenas, obtener su longitud, y copiar fragmentos a otras cadenas. Podemos crear una prueba que se base en realizar varias operaciones con cadenas predeterminadas: concatenación, medir la longitud, fragmentarlas, etc. Sabemos cuál es el resultado correcto de estas operaciones y podemos comprobar la salida de esta rutina. Cada vez que ejecutamos la prueba, la clase obtiene como entradas estas cadenas conocidas y comprueba que las operaciones realizadas han sido correctas.

Este tipo de tests son muy útiles para asegurar que a medida que el software va evolucionando no se rompe la funcionalidad básica del mismo.

3.5.3. Unidades de comprobación

Cuando trabajamos en proyectos de una cierta dimensión, necesitamos tener procedimientos que aseguren la correcta funcionalidad de los diferentes componentes del software, y muy especialmente, de los que forman la base de nuestro sistema. La técnica de las unidades de comprobación, *unit testing* en inglés, se basa en la comprobación sistemática de clases o rutinas de un programa utilizando unos datos de entrada y comprobando que los resultados generados son los esperados. Hoy en día, las unidades de comprobación son la técnica *black boxing* más extendida.

ANOTACIONES

Una unidad de prueba bien diseñada debe cumplir los siguientes requisitos:

- **Debe ejecutarse sin atención del usuario (desatendida)**. Una unidad de pruebas debe poder ser ejecutada sin ninguna intervención del usuario: ni en la introducción de los datos ni en la comprobación de los resultados que tiene como objetivo determinar si la prueba se ha ejecutado correctamente.

- **Debe ser universal**. Una unidad de pruebas no puede asumir configuraciones particulares o basar la comprobación de resultados en datos que pueden variar de una configuración a otra. Debe ser posible ejecutar la prueba en cualquier sistema que tenga el software que es objeto de la prueba.

- **Debe ser atómica**. Una unidad de prueba debe ser atómica y tener como objetivo comprobar la funcionalidad concreta de un componente, rutina o clase.

Dos de los entornos de comprobación más populares en entornos de software libre con JUnit y NUnit. Ambos entornos proporcionan la infraestructura necesaria para poder integrar las unidades de comprobación como parte de nuestro proceso de pruebas. JUnit está pensado para aplicaciones desarrolladas en entorno Java y NUnit para aplicaciones desarrolladas en entorno .Net. Es habitual el uso del término xUnit para referirse al sistema de comprobación independientemente del lenguaje o entorno que utilizamos.

Ejemplo

El siguiente ejemplo es un fragmento de la unidad de comprobación de la clase *StringFormat* del *namespace System.Drawing* del proyecto Mono que son ejecutados con el sistema Nunit de comprobación de unidades.

```
namespace MonoTests.System.Drawing
{
   [TestFixture]
   public class StringFormatTest
   {
      [Test]
      public void TestClone()
      {
```

ANOTACIONES

```
        StringFormat smf = new StringFormat();
        StringFormat smfclone = (StringFormat) smf.Clone();
        Assert.AreEqual (smf.LineAlignment, smfclone.LineAlignment);
        Assert.AreEqual (smf.FormatFlags, smfclone.FormatFlags);
        Assert.AreEqual (smf.LineAlignment, smfclone.LineAlignment);
        Assert.AreEqual (smf.Alignment, smfclone.Alignment);
        Assert.AreEqual (smf.Trimming, smfclone.Trimming);
    }
}
```

Fragmento de la unidad de comprobación de la clase StringFormat del proyecto Mono

El método *TestClone* tiene como objetivo comprobar la funcionalidad del método *Clone* de la clase *StringFormat*. Este método realiza una copia exacta del objeto. Las pruebas se basan en asegurar que las propiedades del objeto han sido copiadas correctamente.

3.6. Sistemas de control de errores

Los sistemas de control de errores son herramientas colaborativas muy dinámicas que proveen el soporte necesario para consolidar una pieza clave en la gestión del control de calidad: el almacenamiento, seguimiento y clasificación de los errores de una aplicación de forma centralizada. El uso de estos sistemas en la mayoría de proyectos se extiende también a la gestión de mejoras solicitadas por el usuario e ideas para nuevas versiones, e incluso, algunos usuarios los utilizan para tener un control de las llamadas de un soporte técnico, pedidos, o los utilizan como gestores de tareas que hay que realizar.

3.6.1. Reportado en errores

Cuando un usuario encuentra un error, una buena forma de garantizar que será buen candidato a ser corregido es escribir un informe de error lo más preciso y detallado posible.

ANOTACIONES

Independientemente de cuál sea el sistema de control de errores que utilicemos, los siguientes consejos reflejan las buenas prácticas en el reporte de errores:

- **Comprobar que el error no ha sido reportado anteriormente**. Suele ocurrir con frecuencia que errores que los usuarios encuentran a menudo son reportados en más de una ocasión. Esto causa que el número de errores totales pendientes de corregir vaya aumentando y que el número total de errores registrados no refleje la realidad. Este tipo de errores se llaman *duplicados* y suelen ser eliminados tan pronto como son detectados. Antes de reportar un nuevo error, es importante hacer una búsqueda en el sistema de control de errores para comprobar que no ha sido reportado anteriormente.

- **Dar un buen título al informe de error**. El título del informe de error debe ser muy descriptivo, ya que esto ayudará a los desarrolladores a poder valorar el error cuando hagan listados de los mismos, y además, ayudará a otros usuarios a encontrar errores ya reportados y evitar reportar posibles duplicados.

  Por ejemplo, un título del tipo "El programa se cuelga" sería un ejemplo de un mal título para describir un error por ser muy poco descriptivo. Sin embargo, "El programa se cuelga en la previsualización de imágenes" es mucho más preciso y descriptivo.

- **Dar una descripción detallada y paso a paso de cómo reproducir el error**. Es importarte que el error sea determinista y que podamos dar una guía paso a paso de cómo reproducirlo para que la persona que debe corregirlo pueda hacerlo fácilmente. Siempre debemos tener presente que la claridad en la descripción del error facilita el trabajo de todos los involucrados en el proceso.

- **Dar la máxima información sobre nuestra configuración y sus particularidades**. La mayoría de sistemas de reporte de error disponen de campos que permiten especificar nuestra configuración (tipo de ordenador, sistema operativo, etc.). Debemos dar la máxima información sobre las particularidades de nuestra configuración,

ya que estos dos datos a menudo son imprescindibles para solucionar un error.

- **Reportar un único error por informe**. Cada informe de error que efectuemos debe contener la descripción de un único error. Esto facilita que el error pueda ser tratado como una unidad por el equipo de desarrollo. Muy a menudo diferentes módulos de un programa están bajo la responsabilidad de diferentes desarrolladores.

Siguiendo todos estos consejos se mejora la claridad de nuestros informes de error y nos ayuda a alcanzar el objetivo final de cualquier reporte de error, que es conseguir que éste pueda ser solucionado con la mayor rapidez y el menor coste posible.

3.6.2. Anatomía de un informe de error

Cualquiera que sea el sistema de control de errores que usemos, todos los sistemas detallan el error con una serie de campos que permiten definirlo y clasificarlo de forma precisa. A continuación describimos los principales campos que forman parte de un informe de error:

- **Identificador**. Código único que identifica el informe de error de forma inequívoca y que asigna el sistema de forma automática. Normalmente, los identificadores suelen ser numéricos, permitiéndonos referirnos al error por su número, como por ejemplo el error 3210.

- **Título**. Frase de aproximadamente unos 50 caracteres que describe el error de forma sintética y le da un título.

- **Informador**. Persona que envía el informe de error. Normalmente, un usuario registrado en el sistema de control de errores.

- **Fecha de entrada**. Fecha en la que el error fue registrado en el sistema de control de errores.

ANOTACIONES

ANOTACIONES

- **Estado**. Situación en la que se encuentra el error. En el apartado de ciclo de vida de un error, veremos los diferentes estados por los cuales puede pasar un error.

- **Gravedad**. Indica la gravedad del error dentro del proyecto. Existen diferentes categorías definidas que abarcan desde un error trivial a un error que puede ser considerado grave.

- **Palabras clave**. Conjunto de palabras clave que podemos usar para definir el error. Por ejemplo, podemos usar *crash* para indicar que es un error que tiene como consecuencia que el programa se cuelgue. Esto es muy útil porque luego permite a los desarrolladores hacer listados o búsquedas por estas palabras clave.

- **Entorno**. Normalmente existen uno o más campos que permiten definir el entorno y configuración del usuario donde se produce el error. Algunos sistemas, como Bugzilla, tienen campos que permiten especificar la información por separado, como por ejemplo: sistema operativo, arquitectura del sistema, etc.

- **Descripción**. Una descripción del error y cómo reproducirlo. Algunos sistemas permiten que diferentes usuarios puedan ir ampliando la información del error a medida que el ciclo de vida del error avanza con detalles relacionados con su posible solución, su implicación para otros errores, o simplemente para concluir que el error ha sido solucionado.

Éstos son los campos que podemos considerar los mínimos comunes a todos los sistemas de control de errores. Cada sistema añade campos adicionales propios para ampliar esta información básica o permitir realizar funciones avanzadas propias.

3.6.3. Ciclo de vida de un error

El ciclo de vida de un error comienza cuando es reportado y finaliza cuando el error se soluciona. Entre estos extremos del ciclo de

vida existen una serie de estados por los cuales va pasando el error.

Aunque no hay un estándar definido entre los diferentes sistemas de control de errores, utilizaremos la terminología que utiliza el sistema Bugzilla, ya que es el más extendido dentro del mundo del software libre.

Éstos son los principales estados en los que se puede encontrar un error durante su ciclo de vida con Bugzilla:

- **Sin confirmar**. Se produce cuando el informe de error ha sido introducido en el sistema. Es el estado inicial por el que pasa cualquier error.
- **Nuevo**. El error estaba en estado "Sin Confirmar" y se ha comprobado su validez y ha pasado al estado "Nuevo", que representa que ha sido aceptado formalmente como error.
- **Asignado**. El error ha sido asignado a un desarrollador para que lo solucione. Ahora debemos esperar a que el desarrollador tenga tiempo para poder evaluar el error y solucionarlo.
- **Corregido**. El error ha sido corregido y debería estar solucionado.
- **Cerrado**. El error ha sido corregido y ha confirmado la solución como correcta. Éste es el fin de ciclo de vida del error.
- **Inválido**. El error no era un error válido o simplemente no era un error. Por ejemplo, cuando se describe una funcionalidad que es correcta.

A continuación vamos a ver un esquema que ilustra los principales estados, Bugzilla y otros sistemas disponen de estados adicionales en los que puede estar un error durante su ciclo de vida.

ANOTACIONES

Figura 1. Principales estados en los que puede estar un error durante su ciclo de vida

Podemos observar que cuando un error entra en el sistema queda clasificado como "Sin Confirmar". Una vez es revisado, puede descartarse como error y marcarse como "Inválido" para luego "Cerrar" el error. Si el error es aceptado como "Nuevo", el siguiente paso es asignarlo a un desarrollador, que es un usuario en el sistema de control de errores. Una vez está asignado, el error es corregido, y una vez verificado, es cerrado.

3.6.4. Bugzilla

Un aspecto central en cualquier proyecto de software es la gestión y el seguimiento de los errores. Cuando Netscape en 1998 liberó el código de Mozilla, se encontró con la necesidad de tener una aplicación de gestión de errores vía web que permitiera la interacción entre usuarios y desarrolladores. Decidieron adaptar la aplicación que usaban internamente en Netscape a las necesidades de un proyecto abierto y así nació Bugzilla. Inicialmente, fue el sistema de gestión y seguimiento de errores del proyecto Mozilla, pero con el tiempo ha sido adoptado por muchos proyectos libres, incluyendo KDE y GNOME, entre otros. Bugzilla permite a los usuarios enviar errores facilitando la clasificación del error, su asignación a un desarrollador para que lo resuelva y todo el seguimiento de las incidencias relacionadas.

Lectura recomendada

http://www.bugzilla.org/

ANOTACIONES

Entre las características que provee Bugzilla, destacan las siguientes:

- **Interfaz web**. Una completa interfaz web que permite que cualquier persona con un simple navegador pueda enviarnos un error o administrar el sistema.

- **Entorno colaborativo**. Cada reporte de error se convierte en un hilo de discusión donde usuarios, probadores y desarrolladores pueden discutir sobre las implicaciones de un error, su posible origen o cómo corregirlo.

- **Notificación por correo**. Bugzilla permite notificar por correo a los usuarios de diferentes eventos, como por ejemplo informar a los usuarios involucrados en un error concreto de cambios en el mismo.

- **Sistema de búsquedas**. El sistema de búsquedas es muy completo, permitiéndonos buscar errores por su tipo, por quién informó de ellos, por la prioridad que tienen o por la plataforma en que han surgido.

- **Informes**. Tiene integrado un completo sistema que permite generar informes sobre el estado de los errores del proyecto.

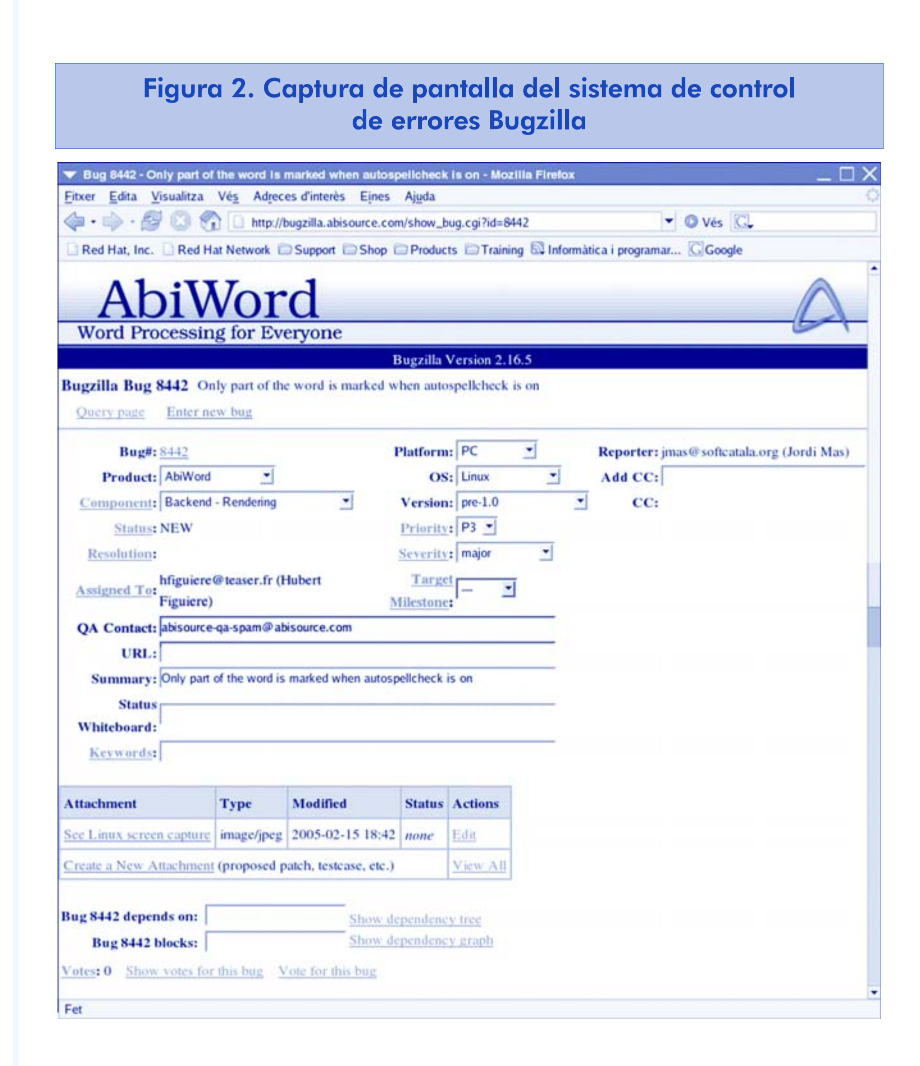
- **Votaciones**. Bugzilla permite la votación de los errores pudiéndose así establecer prioridades para su resolución basándose en los votos que han recibido de usuarios.

- **Seguro**. Contempla registro de usuarios, diferentes niveles de seguridad y una arquitectura diseñada para preservar la privacidad de sus usuarios.

ANOTACIONES

ANOTACIONES

Figura 2. Captura de pantalla del sistema de control de errores Bugzilla

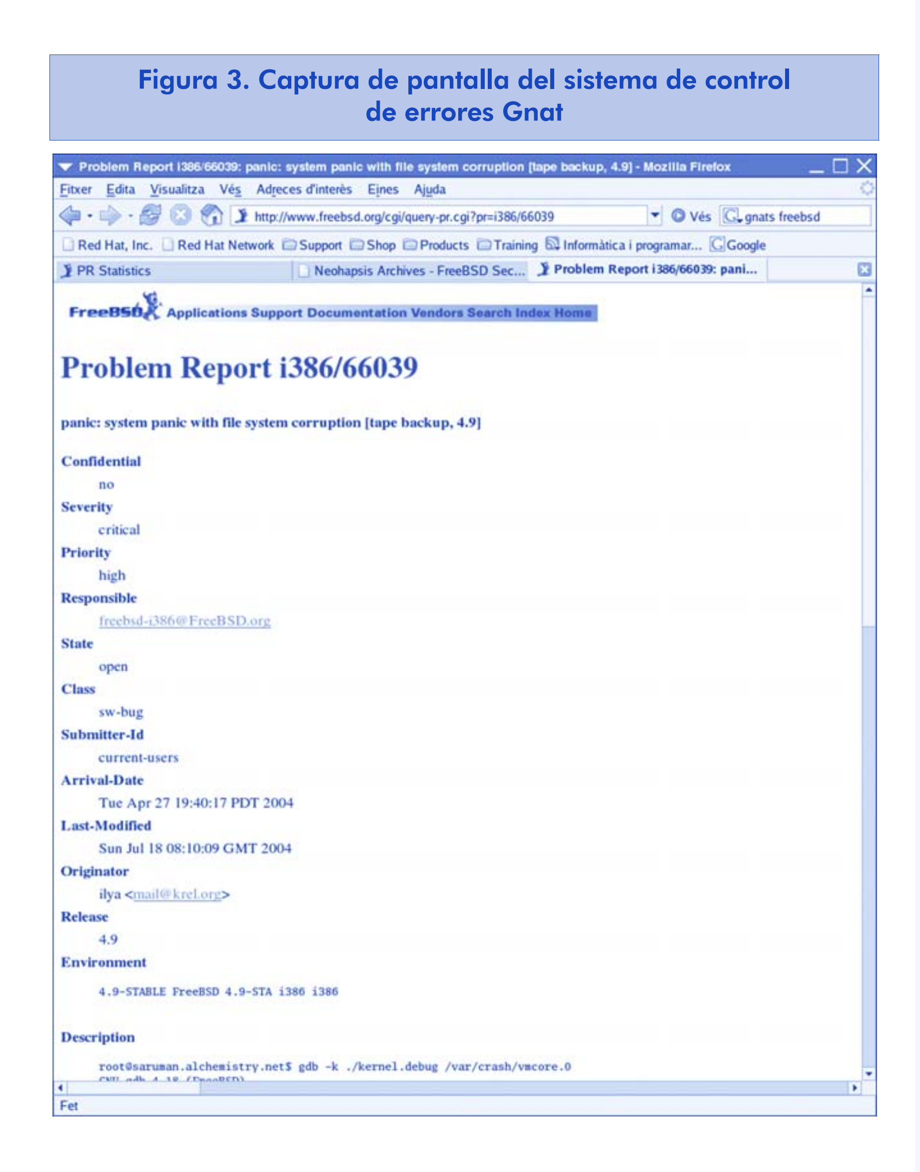


3.6.5. Gnats

Gnats (*GNU problem report management system*) es la herramienta de gestión de errores para entornos Unix desarrollada por la Free Software Foundation a principios de los años noventa. Es más antigua que Bugzilla. Gnats es usado por proyectos como FreeBSD o Apache, y lógicamente, por los proyectos impulsados por la propia Free Software Foundation.

El núcleo de Gnats es un conjunto de herramientas de línea de comandos que exponen toda la funcionalidad del sistema. Tiene todas las ventajas y flexibilidad de las herramientas GNU, pero su uso puede resultar difícil a los que no estén familiarizados con este tipo de herramientas. Existe también una interfaz web que fue desarrollada posteriormente y es el *front-end* más usado hoy en día.

Figura 3. Captura de pantalla del sistema de control de errores Gnat



3.7. Conclusiones

Durante este capítulo hemos constatado la necesidad de integrar los procesos de control de calidad y pruebas dentro del ciclo de desarrollo de un programa desde el inicio del mismo. Hemos visto los principios que rigen la comprobación del software, la escritura de un buen informe de error y de una unidad de comprobación.

También hemos conseguido una visión práctica de los sistemas de control de calidad y pruebas en el caso particular, tanto de prácticas como de aplicaciones del software libre y hemos visto cuáles son los principales principios, técnicas y aplicaciones que se utilizan para garantizar la calidad de los productos libres.

3.8. Otras fuentes de referencia e información

Lewis, W. (2000). *Software Testing and Continuous Quality Improvement*. CRC Press LLC.

Meyers, G. (2004). *The Art of Software Testing*. John Wiley & Sons.

ANOTACIONES

4. Construcción de software en entorno GNU

4.1. Introducción

El sistema operativo GNU/Linux ha supuesto una auténtica revolución en el mundo del software libre. Como consecuencia de su expansión, el desarrollo de software en entornos GNU se ha multiplicado por diez en el último año (2004). Gran cantidad de compañías que desarrollaban sus productos en entorno propietario migran estos a plataformas libres, lo que demanda un gran número de programadores en un entorno muy distinto al habitual en el software propietario.

4.2. Objetivos

Este capítulo pretende al programador lo siguiente:

- Poder introducirse en el entorno de desarrollo GNU guiándole no sólo en la instalación de los componentes necesarios para comenzar a desarrollar, sino en la creación de los ficheros de configuración y en el reconocimiento de los generados tras una compilación, a mantener estos ficheros, a documentar el código y sus cambios, así como, a usar las librerías y las utilidades relacionadas con éstas.

4.3. Instalando todo lo necesario. Autotools

Las autotools son un conjunto de utilidades para automatizar la compilación de programas en diferentes entornos. Permiten al desarrollador especificar unas reglas generales para la compilación y que el resto de la configuración sea detectado en la máquina destino, simplificando la tarea de portabilidad entre sistemas y en general la creación de ficheros `Makefile` de un proyecto.

En realidad se componen de dos paquetes:

http://directory.fsf.org/GNU/autoconf.html
http://directory.fsf.org/GNU/automake.html

Ambos están interrelacionados entre sí y necesitan un intérprete de Perl y de macros m4.

La estructura de funcionamiento es la siguiente:

1) `automake` traduce un fichero de configuración Makefile.am y genera un `Makefile.in`.

2) `autoconf` procesa un `configure.in` (o `configure.ac`) y genera `./configure`.

3) En la máquina destino se ejecuta `./configure` que genera los `Makefile` y opcionalmente otros archivos, posiblemente en todos los directorios que haga falta.

4.3.1. Fichero de configuración de usuario (*input*)

Un proyecto de software constará de una serie de ficheros con código fuente y una serie de pasos a dar para convertirlos en programa ejecutable.

Estos pasos, bajo autotools, se dividen en dos partes principales:

1) Por un lado, un fichero `Makefile.am` a partir del cual se creará el `Makefile` final siguiendo el cual será compilado el código fuente.

2) Por otro, un fichero `configure.in` o `configure.ac` que servirá de plantilla para crear un fichero de script de shell con los pasos necesarios para detectar y establecer la configuración necesaria para que el programa sea compilado e instalado sin problemas. También será el encargado de crear los `Makefile` necesarios siguiendo el `Makefile.am` inicial.

ANOTACIONES

Sin embargo, incluso estos pasos se pueden automatizar con la utilidad autoscan, que generará un `configure.ac` preliminar, con el nombre de `configure.scan`, basándose en los ficheros de código fuente que encuentre en el directorio (y subdirectorios) donde se ejecute.

Veamos este funcionamiento con un programa de ejemplo extremadamente simple:

nul.c

```
void main() {}
```

Creamos un `Makefile.am` sencillo:

Makefile.am

```
bin_PROGRAMS = nul
nul_SOURCES = nul.c
```

En este ejemplo sólo crearemos un binario `nul` a partir del fichero de código fuente `nul.c`.

Simplemente ejecutando `autoscan` obtendremos un fichero `configure.ac` base:

configure.scan

```
AC_PREREQ(2.59)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([nul.c])
AC_CONFIG_HEADER([config.h])

# Checks for programs.
AC_PROG_CC

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Este fichero es recomendable editarlo para adecuarlo al proyecto que estamos desarrollando:

configure.ac

```
AC_INIT(nul, 0.1)
AM_INIT_AUTOMAKE(nul, 0.1)
AC_CONFIG_SRCDIR([nul.c])

# Checks for programs.
AC_PROG_CC

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

ANOTACIONES

El significado de este fichero sería:

- Inicializar autoconf (con el nombre y versión del programa a compilar).
- Inicializar automake (otra vez con nombre y versión).
- Establecer el directorio orígen como el de nul.c.
- Buscar un compilador de lenguaje C.
- Establecer ficheros de configuración Makefile.
- Generar los ficheros de salida.

4.3.2. Ficheros generados (*output*)

Antes de poder generar los ficheros de salida definitivos, deberemos pre-configurar las macros que usarán las autotools, y ya podremos ejecutar autoconf:

```
$ aclocal
$ autoconf
$ ls -l
-rw-r--r--  1 user user  37129 nov 17 18:03 aclocal.m4
drwxr-xr-x  2 user user   4096 nov 17 18:03 autom4te.cache/
-rwxr-xr-x  1 user user 103923 nov 17 18:03 configure*
-rw-r--r--  1 user user    145 nov 17 18:03 configure.ac
-rw-r--r--  1 user user     39 nov 17 18:03 Makefile.am
-rw-r--r--  1 user user     15 nov 17 18:03 nul.c
```

Podría parecer que ya está creado ./configure, sin embargo todavía tenemos que preparar las macros de automake para que funcione:

```
$ automake --add-missing
automake: configure.ac: installing `./install-sh'
automake: configure.ac: installing `./mkinstalldirs'
automake: configure.ac: installing `./missing'
automake: configure.ac: installing `./config.guess'
automake: configure.ac: installing `./config.sub'
automake: Makefile.am: installing `./INSTALL'
automake: Makefile.am: required file `./NEWS' not found
automake: Makefile.am: required file `./README' not found
automake: Makefile.am: installing `./COPYING'
automake: Makefile.am: required file `./AUTHORS' not found
automake: Makefile.am: required file `./ChangeLog' not found
```

ANOTACIONES

Como se puede observar, `automake` ha instalado una serie de scripts necesarios para ejecutar `./configure` en conjunción con `automake`, y además ha creado dos ficheros `INSTALL` y `COPYING`, informando de que faltan otros cuatro.

Veamos lo que son cada uno de estos ficheros:

- `INSTALL`: este fichero describe el proceso de instalación. Automake crea por defecto un fichero con una guía de instalación básica, pero será recomendable editarlo para adecuarlo a las necesidades de cada proyecto en concreto.
- `COPYING`: especifica la licencia bajo la que se permite o restringe la realización de copias del software y código fuente del proyecto.
- `NEWS`: una lista de las modificaciones ("novedades") del proyecto. Suele ir ordenado con las últimas entradas al principio y de forma decreciente según las versiones.
- `README`: este fichero suele ser históricamente el primero en ser mirado por el usuario. Suele incluir una descripción corta y tal vez recomendaciones sobre el uso y otros detalles relativos al paquete. También advertencias que puedan evitar problemas al usuario.
- `AUTHORS`: una lista de las personas que han intervenido en el desarrollo del proyecto. Suelen incluirse las direcciones de correo y posiblemente otras formas de contacto, tanto para aspectos legales de propiedad intelectual como para consultas, sugerencias e informes de bugs (fallos), aparte del merecido reconocimiento de cada uno.
- `ChangeLog`: uno de los ficheros más importantes, que permitirá conocer los cambios, actualizaciones y estado en general del proyecto de software (información adicional en sección más adelante).

Siguiendo con el ejemplo simple, por ahora rellenaremos estos ficheros con datos simples:

```
$ echo '2004-11-17: Proyecto creado' > NEWS
$ echo 'No hace nada. Sintaxis: nul' > README
$ echo 'David Aycart <david@aycart.com>' > AUTHORS
$ cat <<END > ChangeLog
2004-11-17  David Aycart  <david@aycart.com>
       * nul.c: creado
END
```

ANOTACIONES

Ahora ya podremos ejecutar `automake` sin que dé error:

```
$ automake --add-missing
$ ls
aclocal.m4       config.guess@  configure.ac  Makefile.am    NEWS
AUTHORS          config.log     COPYING@      Makefile.in    nul.c
autom4te.cache/  config.sub@    INSTALL@      missing@       README
ChangeLog        configure*     install-sh@   mkinstalldirs@
```

En este punto ya sería posible comprimir esta estructura de directorio y distribuirla como paquete `.tar`, o añadir los ficheros de control necesarios y crear un paquete de código fuente `rpm`, `deb`, o cualquier otro.

Para compilarlo a partir de este punto sólo hacen falta ejecutar `./configure` y después `make all`:

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking whether make sets $(MAKE)... yes
checking for working aclocal-1.4... found
checking for working autoconf... found
checking for working automake-1.4... found
checking for working autoheader... found
checking for working makeinfo... found
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ANSI C... none needed
configure: creating ./config.status
config.status: creating Makefile

$ make all
gcc -DPACKAGE_NAME=\"nul\" -DPACKAGE_TARNAME=\"nul\" -DPACKAGE_VERSION=\"0.1\"-DPACKAGE_STRING=\"nul\
0.1\" -DPACKAGE_BUGREPORT=\"\" -DPACKAGE=\"nul\" -DVERSION=\"0.1\"  -I. -I.      -g -O2 -c nul.c
nul.c: In function `main':
nul.c:1: warning: return type of 'main' is not `int'
gcc  -g -O2  -o nul  nul.o

$ ls -l nul
-rwxr-xr-x  1 user user 7893 nov 17 17:03 nul
```

4.3.3. Como mantener ficheros de entrada

Si se edita alguno de los ficheros de entrada de las autotools, esto es `configure.ac` o `Makefile.am`, será necesario actualizar los ficheros generados por `autoconf` y `automake`.

Una forma es crearse un script que repita los pasos que hemos dado anteriormente para crear los ficheros. En este caso las autotools ya traen un programa que hace exactamente esto, llamado `autoreconf`. Exactamente lo que hace es ejecutar una secuencia estándar de las utilidades de autotools que normalmente es suficiente para actualizar los ficheros.

De hecho, en proyectos simples como el del ejemplo anterior (`nul.c`), sería posible ejecutar `autoreconf` directamente después de crear `configure.ac` y `Makefile.am`. En ese caso, añadiendo los ficheros de documentación de cuya falta informaría autoreconf (al ejecutar `automake --add-missing`) y ejecutándolo una vez más.

4.3.4. Empaquetar ficheros *output*

Una práctica común en los paquetes de código fuente distribuidos junto a las mayores distribuciones, es la inclusión de todos los ficheros generados de tal forma que sea suficiente ejecutar `./configure` y `make all` o `make install` para que el programa sea totalmente funcional en el sistema destino.

Esto tiene dos implicaciones, una buena y otra no tanto.

Por un lado, a un usuario novel (o no tan novel) le facilita mucho la tarea de pasar parches y poder disfrutar del programa corregido en el menor tiempo posible sin necesitar mayores conocimientos sobre el sistema.

Por otro, si estos ficheros fuente se intentasen compilar en otro sistema sería posible que el fichero `./configure` no fuese completamente compatible con las autotools instaladas, por lo que haría falta re-generarlo desde cero.

ANOTACIONES

Opinamos que se deberían distribuir ambos tipos de ficheros, tanto los originales `configure.ac` y `Makefile.am` como sus derivados hasta un conjunto directamente ejecutable en una plataforma preferida, especificando claramente cuál es esta plataforma (por ejemplo, una distribución y versión determinada), aunque tal vez también sería interesante incluir un listado de los ficheros originales antes de ejecutar las autotools, o incluso un script que permitiese quitar los ficheros generados para satisfacer todas las posibles necesidades.

4.3.5. Changelogs y documentación

Un aspecto que no se debe descuidar es la correcta documentación de los proyectos de software. Aún cuando esto es importante incluso en un entorno de programación "artístico" donde una sola persona desarrolle y mantenga todo el código, este aspecto se vuelve crucial en los desarrollos GNU, donde el mismo código normalmente será analizado y mejorado por diferentes desarrolladores con poco o incluso ningún contacto previo entre sí, sea por motivos geográficos, temporales u otros.

Debido a la relevancia de la documentación, y a la facilidad con la que puede olvidarse incluir, es por lo que automake comprueba su presencia antes de dar por válido el conjunto de ficheros.

El fichero de documentación más relevante, de referencia tanto para usuarios noveles, avanzados como para administradores, es el `ChangeLog`. De ahí que el formato tenga una estructura bastante definida y comunmente aceptada.

El fichero ChangeLog consta de una serie de "entradas", en orden de fechas o versiones decrecientes (lo más nuevo primero), cada una de las cuales comienza con una línea definiendo la fecha, persona y e-mail en que se realizan los cambios, seguida de una lista de cambios separados por líneas en blanco y normalmente con el margen ligeramente desplazado y precedidos de asterisco (*).

Cada cambio suele incluir una referencia del módulo, archivo o funcionalidad cambiada o actualizada, y una descripción corta del cambio en

sí, opcionalmente seguida de un código identificativo (por ejemplo si se trata de una corrección de un bug, el número del bug).

Los cambios que se realicen sobre elementos relacionados pueden incluirse dentro de un solo cambio o detallarse en líneas distintas, sin embargo en el segundo caso pueden no ir separadas por una línea en blanco.

Por ahora veamos unos ejemplos.

Para empezar, un trozo del fichero ./kdb/ChangeLog del kernel:

```
2004-08-14 Keith Owens  <kaos@sgi.com>
       * kdb v4.4-2.6.8-common-1.
2004-08-12 Keith Owens  <kaos@sgi.com>
       * kdb v4.4-2.6.8-rc4-common-1.
2004-08-05 Keith Owens  <kaos@sgi.com>
       * Mark kdb_initcall as __attribute_used__ for newer gcc.
       * kdb v4.4-2.6.8-rc3-common-2.
```

O del fichero ChangeLog de gcc:

```
2004-07-01  Release Manager
       * GCC 3.4.1 released.
2004-06-28  Neil Booth  <neil@duron.akihabara.co.uk>
        PR preprocessor/16192
        PR preprocessor/15913
        PR preprocessor/15572
        * cppexp.c (_cpp_parse_expr): Handle remaining cases where an
        expression is missing.
        * cppinit.c (post_options): Traditional cpp doesn't do // comments.
        * doc/cpp.texi: Don't document what we do for ill-formed expressions.
        * doc/cppopts.texi: Clarify processing of command-line defines.

2004-06-28  Richard Sandiford  <rsandifo@redhat.com>

        PR target/16176
        * config/mips/mips.c (mips_expand_unaligned_load): Use a temporary
        register for the destination of the lwl or ldl.
```

Otros registros de cambios siguen un formato lógico parecido, aunque no directamente compatible, como por ejemplo el fichero CHANGES de Apache 2.0.52:

Lectura recomendada

Una descripción más detallada puede encontrarse en la dirección siguiente:

http://www.gnu.org/prep/standards/html_node/Change-Logs.html#Change-Logs

ANOTACIONES

```
Changes with Apache 2.0.52
*) Use HTML 2.0 <hr> for error pages. PR 30732 [André Malo]

*) Fix the global mutex crash when the global mutex is never allocatec
   due to disabled/empty caches. [Jess Holle <jessh ptc.com>]

*) Fix a segfault in the LDAP cache when it is configured switched
   off. [Jess Holle <jessh ptc.com>]

*) SECURITY: CAN-2004-0811 (cve.mitre.org)
   Fix merging of the Satisfy directive, which was applied to
   the surrounding context and could allow access despite configured
   authentication.  PR 31315.  [Rici Lake <rici ricilake.net>]
```

4.3.6. Creación de configure.in

Una duda que puede plantearse al definir el fichero `configure.in` es la de qué poner dentro. En concreto, qué chequeos deberían de realizarse para que el programa funcionase correctamente, sin excederse en chequeos que puedan fallar en el sistema destino sin que influyesen en la compilación, o simplemente en qué orden realizarlos.

4.3.7. ¿Qué sigifica portabilidad?

Un programa idealmente portable es aquél que un usuario pueda hacer funcionar sin mayor esfuerzo en cualquier plataforma y sistema posible. De forma ideal esto supondría que sólo tuviese que ejecutar un comando para tener el programa instalado y funcionando.

Este sería un caso ideal, que sería trivial de resolver si todos los sistemas y plataformas fuesen idénticos entre sí. Sin embargo, en la realidad esto nunca se cumple, e incluso no sería deseable que se cumpliese. Los sistemas difieren entre sí, las plataformas difieren, incluso dentro de una misma plataforma y sistema difieren los programas, utilidades y las versiones instaladas de cada uno.

Por lo tanto, para ser portable, un programa debe ser capaz de adaptarse al entorno donde vaya a ejecutarse.

Con el fin de que estas adaptaciones no se hagan imposibles debido a la existencia de infinitas variables, hay establecida una serie

ANOTACIONES

de estándares, convenciones y costumbres que facilitarán al programa tanto la identificación del entorno donde va a ejecutarse como la adaptación al mismo.

En el caso concreto de las autotools se suele hablar de sistemas "tipo Unix", tales como Linux, BSD, SunOS, Solaris, etc. además de las distintas variantes de cada uno. Incluso se puede pensar en compatibilidad con sistemas Windows, MacOS u otros.

4.3.8. Introducción al sh portable

La herramienta más importante para la instalación de un programa es el programa de línea de comando en sí. Este será el encargado de servir de base al resto de comandos ejecutados durante el proceso de instalación.

Para evitar problemas de compatibilidad entre distintas implementaciones de sh, especialmente aquellas carentes de algunas funcionalidades extendidas, las autotools intentan generar un fichero `./configure` con el conjunto de funcionalidades de sh más compatible posible. Esto puede llevar a que el código resultante no sea fácilmente legible para alguien acostumbrado a las implementaciones más difundidas.

Por otro lado, la programación bajo `sh` se sirve en gran medida de otros programas "de apoyo", que también tienen distintas implementaciones y no siempre se comportan de una manera exactamente igual a la esperada.

Debido a esto los programas GNU suelen incluir una sección en su documentación (man) detalando las opciones que soportan compatibles con POSIX, el estándar de compatibilidad entre sistemas operativos. Siguiendo estas opciones soportadas y evitando las demás, es bastante probable que un script sea altamente portable.

Lectura recomendada

El estándar POSIX se puede consultar en la dirección siguiente:

http://www.unix.org/version3/ieee_std.html

De especial interés será la sección "Shell Command Language".

4.3.9. Chequeos necesarios

Gracias a autoscan hemos podido ver una configuración básica compatible con el proyecto. Sin embargo, muchas veces no es suficiente

ANOTACIONES

ANOTACIONES

con esta configuración básica y cada proyecto puede tener sus peculiaridades que deban ser tomadas en cuenta a la hora de realizar los chequeos.

Hay que tener en mente que el fichero configure.am es interpretado como fichero de script con "macros intercaladas" que serán expandidas por autoconf (usando m4). Por lo tanto se puede incluir en él cualquier trozo de código de script shell para realizar comprobaciones adicionales, o condicionar la realización de comprobaciones a distintos aspectos.

La secuencia principal recomendada suele ser la siguiente:

1) **Inicialización**. Aquí se incluye el código de inicio de las distintas comprobaciones, como puede ser `AC_INIT` o `AM_INIT_AUTOMAKE`.

2) **Configuración**. Se establecen distintas opciones como `AC_CONFIG_SRCDIR` o `AC_CONFIG_HEADER`.

3) **Programas**. Aquí se comprueba la presencia de programas necesarios para la compilación o la ejecución del mismo proceso de configuración. En caso de estar presentes varios posibles, también se suele elegir el más adecuado.

4) **Librerías**. Se comprueba la existencia de las librerías necesarias para compilar el programa. Esto es recomendable hacerlo antes de los siguientes pasos dado que en algunos se comprueba la presencia de funcionalidades intentando compilar programas de ejemplo que las utilicen.

5) **Encabezados**. Esto se refiere a los encabezados de librerías y módulos instalados en el sistema para los que también estén instalados los encabezados de compilación, necesarios para compilar el programa.

6) **Typedefs y estructuras**. Se realizan comprobaciones de existencia de typedefs y estructuras en los encabezados detectados en el paso anterior.

7) **Funciones**. Esta comprobación es la realmente importante, que detectará la presencia de funciones utilizadas en el programa en

función de los encabezados y librerías presentes, comprobando la posibilidad de usarlas aplicando los typedefs y estructuras encontrados.

8) **Salida**. Este es el paso final que creará los ficheros de salida en función de las opciones detectadas.

4.4. Introducción a GNU Automake

Automake es la utilidad encargada de crear los ficheros `Makefile.in` en base a una definición simple de lo que hace falta hacer y cumpliendo con los estándares de creación de ficheros `Makefile` de GNU.

Por un lado esto facilita la creación de los `Makefile` al no tener que especificar en cada uno todos los pasos que se deben dar, simplificando su creación.

Por otro, al crear los ficheros `Makefile` con un formato uniforme, será más fácil procesarlos con utilidades estándar compatibles con dicho formato.

Al mismo tiempo, dada la gran popularidad de estos ficheros `Makefile`, necesarios para la compilación de prácticamente todos los programas, una gran cantidad de usuarios es capaz de aportar información sobre cualquier tipo de incompatibilidad, error, u optimización que se deba corregir o incorporar a `automake`.

4.4.1. Principios generales de automake

Como ya se ha mencionado, `automake` se encarga de convertir `Makefile.am` en `Makefile.in` que posteriormente será usado por `./configure` como una plantilla para los ficheros `Makefile` finales.

Por lo tanto, los ficheros `Makefile.am` se pueden entender como ficheros `Makefile` normales con macros.

Un aspecto diferente respecto a los `Makefile` son los comentarios internos de automake, que serán descartados al crear el `Makefile.in`, y que empezarán con '`##`' en vez de '`#`':

Lectura recomendada

Una referencia completa de las opciones de configuración se puede encontrar en la dirección siguiente:

http://www.gnu.org/software/autoconf/autoconf.html (Online Manual)

Lectura recomendada

Una referencia completa sobre automake se puede encontrar en la dirección siguiente:

http://directory.fsf.org/GNU/automake.html (User manual)

ANOTACIONES

```
## Este es un fichero Makefile.am de ejemplo (esta línea no aparecerá)
```

Aparte de esto, automake soporta líneas condicionales y expansiones de include que se procesarán para generar el fichero `Makefile.in`.

Todas las instrucciones y posibles macros no interpretadas directamente por automake son pasadas tal cual a `Makefile.in`.

4.4.2. Introducción a las primarias

Las variables "primarias" se utilizan indirectamente para derivar de ellas nombres de objetos a partir de ellas.

Por ejemplo `PROGRAMS` es una primaria, y `bin_PROGRAMS` es un nombre derivado de ella:

```
bin_PROGRAMS = nul
```

En este caso se usa el prefijo especial `bin` que indicaría el destino del programa en "`bindir`". Algunos de los destinos posibles son los siguientes:

- `bindir`: donde deben ir los archivos binarios
- `sbindir`: para los archivos binarios de sistemas
- `libexecdir`: programas y demonios ejecutables por otros programas
- `pkglibdir`: librerías no compartidas en el sistema
- `noinst`: archivos que no se instalarán
- `check`: sólo serán compilados con "make check" para comprobaciones
- `man`: páginas de manual

ANOTACIONES

Para indicar el código fuente que formará el destino “nul” especificado antes, usaremos la primaria SOURCES:

```
nul_SOURCES = nul.c
```

Aparte de estas primarias principales, también están disponibles:

- HEADERS: permite listar los encabezados .h de los ficheros.
- SCRIPTS: lista los scripts ejecutables. Serán copiados sin interpretar, aunque con atributos de ejecución.
- DATA: ficheros que serán copiados directamente. Suelen ser contenidos fijos utilizados por el programa, como fuentes de datos o gráficos.
- MANS: se refiere a las páginas de manual. Suelen usarse como man_MANS o man1_MANS, man2_MANS, etc. para permitir la instalación en secciones concretas del manual.

4.4.3. Programas y librerias

Como ya se ha mencionado, los programas se indican bajo la primaria PROGRAMS.

En el caso de las librerías, se aplica la primaria LIBRARIES:

```
lib_LIBRARIES = libtest.a
libtest_a_SOURCES = ltest.c ltest.h
```

Para liberías libtool (ver más adelante) tenemos la primaria LTLIBRARIES:

```
lib_LIBRARIES = libtest.la
libtest_la_SOURCES = ltest.c ltest.h
```

Como se puede apreciar, el punto de “libtest.a” se convierte en un guión bajo para dar lugar al prefijo de objeto “libtest_a”.

ANOTACIONES

4.4.4. Múltiples directorios

Un proyecto mínimamente grande y bien organizado, suele estar dividido en diferentes subdirectorios.

Para procesar estructuras de directorio respecto al actual, automake acepta la opción SUBDIRS:

```
SUBDIRS = src libsrc man
```

La única restricción es que los directorios deben ser descendientes del actual, por lo que si se desea procesar también un segundo nivel como por ejemplo src/main/, se debería incluir este subdirectorio main en el fichero Makefile.am de src:

```
src/Makefile.am
SUBDIRS = . main
```

4.4.5. Cómo probarlo

Una de las características de automake y de los Makefile generados, es la capacidad de realizar tests de funcionamiento del código compilado.

Bajo el destino check de Makefile, invocado con make check, se ejecutarán una serie de programas especificados en la variable TESTS de Makefile.am. En esta variable se pueden especificar tanto objetos fuente como objetos derivados.

Cada programa testeado debe devolver un valor cero para indicar test pasado, o distinto de cero para indicar test fallido. Los tests que no tengan validez en la plataforma o entorno de compilación, pueden devolver el valor 77 para no ser tenidos en cuenta en el cómputo final.

Igualmente, es posible especificar una lista de tests que se espera que fallen con la variable XFAIL_TESTS, que serán considerados como válidos en el caso inverso que los no especificados. Estos tests también deben incluirse en la variable TESTS para ser ejecutados.

ANOTACIONES

Para especificar un destino común a todos los programas de test, se puede usar el destino check con la primaria `PROGRAMS`:

```
check_PROGRAMS = test1 test2
test1_SOURCES = test1.c
test2_SOURCES = test2.c
```

Así mismo, se pueden ejecutar los tests de DejaGnu con:

```
AUTOMAKE_OPTIONS = dejagnu
```

4.5. La libreria GNU Libtool

Libtool es una herramienta que permite abstraer el proceso de compilación de librerías del formato concreto que estas vayan a tomar forma en la plataforma donde sean compiladas, incluyendo el control de versiones y las dependencias entre librerías.

Esto es especialmente importante en el caso de librerías que puedan ser compartidas en el sistema destino, aunque también es un añadido cómodo para el desarrollo normal de librerías independientemente de la plataforma.

Para conseguirlo, las librerías y los ejecutables se compilan en un formato ejecutable modificado, nombrándose con la extensión `.la` en vez de `.a`.

Además, libtool ha sido pensada de forma que esté integrada con las utilidades autoconf y automake, aunque no sea obligatorio que estas estén instaladas para poder hacer uso de libtool.

4.5.1. Código independiente

Para una librería interna de un programa no es necesario que su código sea independiente de posición (Position Independent Code = PIC). Sin embargo, al crear librerías compartidas entre distintos programas,

Lectura recomendada

Se puede encontrar más información sobre Libtool en la dirección siguiente:

http://www.gnu.org/software/libtool/manual.html

ANOTACIONES

hace falta que la dirección en la que se carga la librería sea alterable, de forma que las direcciones internas deberán ser relativas al punto de inicio, o alteradas en el momento de cargar la librería.

Para crear código independiente, hace falta pasar una serie de opciones al compilador. Esta tarea puede ocultarse usando libtool, que se encargará de todo el proceso de compilado.

Normalmente, el código independente se generará como objetos `.lo`, a diferencia del código normal `.o`.

4.5.2. Librerias compartidas

Por defecto, `libtool` siempre creará librerías compartidas, salvo en las plataformas donde esto no sea posible o que se le requiera explícitamente construir librerías estáticas.

Veamos esto con unos ejemplos.

Para ello vamos a crear una librería básica de ejemplo:

suma.c

```
int suma (int a, int b) {
 return a+b;
}
```

Una compilación normal sin utilizar `libtool` consistiría en:

```
$ gcc -c suma.c
$ ls
suma.c  suma.o
$ ar cru libsuma.a suma.o
$ ranlib libsuma.a
$ ls
libsuma.a  suma.c  suma.o
```

En este caso la librería creada es estática.

ANOTACIONES

Para crear una librería compartida, podríamos recurrir a libtool:

```
$ libtool --mode=compile gcc -c suma.c
mkdir .libs
  gcc -c suma.c  -fPIC -DPIC -o .libs/suma.o
  gcc -c suma.c -o suma.o >/dev/null 2>&1
$ libtool --mode=link gcc -o libsuma.la suma.lo -rpath /usr/local/lib/
rm -fr  .libs/libsuma.a .libs/libsuma.la
gcc -shared  .libs/suma.o   -Wl,-soname -Wl,libsuma.so.0 -o .libs/libsuma.so.0.0.0
(cd .libs && rm -f libsuma.so.0 && ln -s libsuma.so.0.0.0 libsuma.so.0)
(cd .libs && rm -f libsuma.so && ln -s libsuma.so.0.0.0 libsuma.so)
ar cru .libs/libsuma.a .libs/suma.o
ranlib .libs/libsuma.a
creating libsuma.la
(cd .libs && rm -f libsuma.la && ln -s ../libsuma.la libsuma.la)
$ ls
libsuma.la  suma.c  suma.lo  suma.o
```

Con la opcion –rpath especificamos la dirección donde estará instalada la librería.

4.5.3. Librerias estáticas

Para aprovechar la capacidad de interfaz estándar de creación de librerías, libtool también permite crear librerías estáticas sin tener que llamar directamente al compilador, especificando la opción –static en la fase de enlazado:

```
$ libtool --mode=link gcc -static -o libsuma.la suma.lo
rm -fr  .libs/libsuma.a .libs/libsuma.la
ar cru .libs/libsuma.a  suma.o
ranlib .libs/libsuma.a
creating libsuma.la
(cd .libs && rm -f libsuma.la && ln -s ../libsuma.la libsuma.la)
```

De esta forma se creará una librería que contendrá todos los objetos referenciados por la misma.

Para probar el correcto funcionamiento de esta librería, deberemos enlazarla junto a un ejecutable.

ANOTACIONES

Por ejemplo, en este caso usaremos este ejecutable mínimo:

main.c

```
#include <stdio.h>

int suma (int a, int b);

int main (int argc, char *argv[]) {
 printf ("%i", suma (1, 2) );
 exit (0);
}
```

Ahora compilemos este programa incluyendo la librería anterior:

```
$ libtool --mode=link gcc -o suma main.c libsuma.la
gcc -o .libs/suma main.c  ./.libs/libsuma.so -Wl,--rpath -Wl,/usr/local/lib,
creating suma
$ ./suma
3
```

Vemos que el programa funciona perfectamente, llamando la función de la librería.

4.5.4. Enlazar una librería. Dependencias entre librerias

Es bastante común enlazar una librería con otras ya existentes para añadir o agrupar funcionalidad en una sola, o para maximizar las capacidades de portabilidad de un programa al no depender de la existencia o no de librerías en el sistema destino.

`Libtool` usará el sistema soportado en destino para la inter-dependencia de librerías, de manera transparente al usuario.

En caso de no encontrar un medio nativo para resolverlas, `libtool` ofrece una emulación propia.

ANOTACIONES

4.5.5. Usar librerías de conveniencia

Las librerías de conveniencia son librerías parcialmente enlazadas que pueden usarse como agrupaciones intermedias de objetos aunque no sean ni ejecutables ni librerías cargables en sí. Estas librerías se definen simplemente sin especificar ni -static ni -rpath en el momento de llamar libtool como `--mode=compile`, y son usadas sólo para ser enlazadas dentro de otras librerías o ficheros ejecutables.

4.5.6. Instalación de librerias y ejecutables

La instalación de librerías por medio de `libtool` en realidad no supone mayor problema.

```
$ libtool --mode=install cp libsuma.la /usr/local/lib/libsuma.la
cp libsuma.la /usr/local/lib/libsuma.la
cp .libs/libsuma.a /usr/local/lib/libsuma.a
ranlib /usr/local/lib/libsuma.a
```

Para finalizar la instalación usaremos `--mode=finish`:

```
libtool --mode=finish /usr/local/lib
PATH="$PATH:/sbin" ldconfig -n /usr/local/lib
--------------------------------------------------------------------
Libraries have been installed in:
   /usr/local/lib
If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,--rpath -Wl,LIBDIR' linker flag
  - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
--------------------------------------------------------------------
```

Con esto ya podremos usar nuestra librería compartida.

ANOTACIONES

4.5.7. Desinstalación

La desinstalación podemos realizarla por medio de --mode=uninstall con el comando rm:

```
$ libtool --mode=uninstall rm -f /usr/local/lib/libsuma.la
rm -f /usr/local/lib/libsuma.la /usr/local/lib/libsuma.a
rm -f /usr/local/lib/libsuma.la /usr/local/lib/libsuma.so.0.0.0 /usr/local/lib/libsuma.so.0 /usr/
local/lib/libsuma.so /usr/local/lib/libsuma.a
```

De esta forma se asegura que la librería quedará desinstalada independientemente de la plataforma, incluidos cualquier enlace a la misma que pudiese haber sido necesario crear en la instalación.

4.5.8. GNU Libtool, configure.in y Makefile.am

Para utilizar libtool con autoconf y automake sólo hace falta definir su uso en configure.ac.

La macro que nos ofrece autoconf para ello es la AC_PROG_LIBTOOL. Al mismo tiempo, no harán falta comprobaciones adicionales relacionadas con la compilación de librerías, pues de todo ello se encargará libtool:

```
AC_PROG_LIBTOOL
```

Tras esto, sólo tendremos que ejecutar autoconf y automake como de costumbre.

4.5.9. Integración con configure.in, opciones extra y macros para Libtool

La compilación con libtool añade una serie de opciones nuevas a ./configure, entre ellas:

--enable-shared

--disable-shared: permiten activar o desacrivar la compilación de librerías compartidas.

ANOTACIONES

`--enable-static`

`--disable-static`: igualmente para librerías estáticas.

`--with-pic`: fuerza a construir librerías con código independiente (PIC).

Estas opciones tienen sus correspondientes opciones de `configure.ac`:

```
AC_DISABLE_SHARED
```

para desactivar las compilación de librerías compartidas, y

```
AC_DISABLE_STATIC
```

para desactivar las compilación de estáticas.

4.5.10. Integración con Makefile.am, creación de librerias con Automake y linkado contra librerias Libtool

Automake soporta la compilación y enlazado de librerías por medio del destino `lib` y la primaria `LIBRARIES`:

```
lib_LIBRARIES = libsuma.a
libsuma_a_SOURCES = suma.c
```

La construcción de librerías con el uso de libtools, utiliza la primaria `LTLIBRARIES`:

```
lib_LTLIBRARIES = libsuma.la
libsuma_la_SOURCES = suma.c
```

El enlazado contra librerías `libtool` se obtiene por medio de la primaria `LDADD`:

```
bin_PROGRAMS = suma
suma_SOURCES = main.c
suma_LDADD = libsuma.c
```

ANOTACIONES

4.5.11. Utilización de libtoolize

La herramienta que nos ayudará a configurar nuestro proyecto para hacer uso de `libtool`, como añadir y comprobar que está presente el soporte para `libtool`, es libtoolize.

El proceso completo de creación de nuestro proyecto, haciendo uso de `libtool`, puede ser el siguiente.

configure.ac

```
AC_INIT(suma, 0.1)
AM_INIT_AUTOMAKE(suma, 0.1)
AC_CONFIG_SRCDIR([suma.c])

# Checks for programs.
AC_PROG_CC
AC_PROG_LIBTOOL

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Makefile.am

```
bin_PROGRAMS = suma
suma_SOURCES = main.c
suma_LDADD = libsuma.la

lib_LTLIBRARIES = libsuma.la
libsuma_la_SOURCES = suma.c
```

Una vez hechas las modificaciones correspondientes para añadir soporte de libtool y establecer los ficheros fuente y destino a usar en `configure.ac` y `Makefile.am`, comprobemos que el proyecto compila correctamente:

```
$ ls
AUTHORS  ChangeLog  configure.ac  main.c  Makefile.am  NEWS  nul.c  README  suma
$ libtoolize
You should add the contents of `/usr/share/aclocal/libtool.m4' to `aclocal.m4'.
$ aclocal
$ automake --add-missing
automake: configure.ac: installing `./install-sh'
automake: configure.ac: installing `./mkinstalldirs'
automake: configure.ac: installing `./missing'
automake: Makefile.am: installing `./INSTALL'
automake: Makefile.am: installing `./COPYING'
```

ANOTACIONES

```
$ autoconf
$ ./configure
[...]
$ make all
[...]
creating suma
$ ls *suma*
libsuma.la suma*  suma.c  suma.lo  suma.o
$ ./suma
3
```

Como podemos observar, se han creado todos los elementos correctamente y compilado el programa final.

Nota

En algunos casos no hará falta ejecutar `libtoolize` directamente, sino que será llamado por `automake –add-missing` (en este ejemplo habría funcionado correctamente).

4.5.12. Versionado de librerías

La creación de librerías, especialmente las compartidas a ser instaladas en el sistema, requiere una serie de consideraciones adicionales relativas a la versión de las librerías distribuidas.

En el momento de la carga de librerías, será elegida aquella que cumpla una serie de condiciones relativas a la compatibilidad con el programa que la necesite, definidas por medio del número de versión. Este número de versión, aunque puede ser distinto en cada plataforma, es abstraído por `libtool` a un formato unificado visto en función de las interfaces exportadas por la librería. De esta forma se permite que un programa con interfaz compatible pueda cargar una librería posterior a su distribución, que puede tener corregidos fallos o añadida funcionalidad sin perder compatibilidad hacia atrás.

Como interfaz entenderemos, a efectos de versionado, los tipos y nombres de variables globales, las funciones globales (tipos de argumentos, valor de retorno, nombres de funciones), entrada/salida/error estándar y formatos de ficheros, sockets, tuberías, y otras formas de comunicación entre procesos y sus formatos.

Bajo libtool las librerías se entiende que exportan “conjuntos de interfaces” numerados secuencialmente. Al intentar cargarlas, será elegida aquella librería que soporte todos los números de interfaz requeridos por el programa.

Los números de interfaz se establecen por medio de la opción `--version-info`, especificada en el modo de `libtool --mode=link` o en la primaria `LDFLAGS` de `Makefile.am`, con el formato de:

```
actual : revisión : antigüedad
```

- `actual`: establece el número de la interfaz más moderna soportada.
- `revisión`: permite distribuir librerías actualizadas con la misma interfaz.
- `antigüedad`: número de interfaces anteriores soportadas.

El total de interfaces soportadas estará definido por el intervalo:

desde `actual – antigüedad` hasta `actual`

Así, una librería definida como 1:1:0 soportará sólo la interfaz número 1, revisión 1.

Mientras, la librería definida 2:1:1 soportará tanto la interfaz 1 como la 2, de la cual soportará la revisión 1, por lo que podrá ser cargada en lugar de la 1:1:0.

La librería con versión 2:2:1, igualmente podrá ser cargada por los programas que necesiten la 1 o la 2:1:1.

Sin embargo una librería 2:3:0, habrá roto la compatibilidad con los programas para la 1, y sólo será compatible con los que acepten la 2, aunque no con un programa que necesite la 2:4.

4.6. Conclusiones

La preparación de un entorno de desarrollo GNU es muy distinta en la forma, las utilidades y aplicaciones a instalar y el fondo a los entornos propietarios y visuales usados en sistemas operativos Windows.

ANOTACIONES

Estas diferencias se ponen de manifiesto en el uso de herramientas poco comunes en otros sistemas, no solo funcional sino visualmente.

En este capítulo se han definido las tareas, sentado las bases y reconocido el software necesario para configurar nuestro entorno de desarrollo GNU, de forma que podamos avanzar en el uso de IDEs de desarrollo y herramientas visuales en capítulos consecutivos.

ANOTACIONES

5. Control de versiones

5.1. Introducción

Los sistemas de control de versiones funcionan como la columna vertebral permitiendo a grupos de personas trabajar de forma conjunta en el desarrollo de proyectos, frecuentemente a través de Internet. Son sistemas que señalan las diferentes versiones para identificarlas posteriormente, facilitan el trabajo en paralelo de grupos de usuarios, indican la evolución de los diferentes módulos del proyecto, y disponen de un control detallado de los cambios que se han realizado; funciones que son indispensables durante la vida del proyecto. Estos sistemas no sólo tienen aplicación en el desarrollo del software, sino que además son ampliamente utilizados en la creación de documentación, sitios web y en general cualquier proyecto colaborativo que requiera trabajar con equipos de personas de forma concurrente.

En el mundo del software propietario, los sistemas de control de versiones han sido tradicionalmente usados para gestionar equipos de desarrollo en entornos corporativos cerrados. Con la explosión de Internet y el avance del software libre, se diseñaron sistemas abiertos que permitieran trabajar simultáneamente a miles de personas distribuidas en diferentes regiones del mundo en un mismo proyecto a través de Internet.

CVS (*concurrent versions system*) es el programa más utilizado en el mundo del software libre para el control de versiones de software. Basado en el modelo cliente-servidor, existen versiones del programa para multitud de plataformas. Su solidez y su probada eficacia, tanto en grupos pequeños de usuarios como en grandes, le han convertido en la herramienta que utilizan proyectos de software libre de éxito reconocido como Mozilla, OpenOffice.org, KDE o GNOME, por mencionar sólo algunos. Veremos también Subversion, un nuevo sistema de control de versiones con prestaciones superiores a CVS.

ANOTACIONES

Durante este capítulo aprenderemos cómo utilizar CVS y Subversion para facilitar el trabajo entre un grupo de usuarios que trabajan en un mismo proyecto y veremos la utilidad de disponer de un sistema de control de versiones.

5.2. Objetivos

Los materiales didácticos asociados a este capítulo permitirán obtener los siguientes conocimientos:

- Familiarizarse con la parte cliente de CVS y Subversion para poder seguir la evolución de un proyecto de software donde participen diversos desarrolladores.
- Cómo enviar cambios y correcciones a través de CVS o por correo electrónico usando los comandos *diff* y *patch*.
- Cómo instalar y administrar un servidor CVS propio donde publicar los proyectos y compartirlos con otros usuarios.
- El funcionamiento de los sistemas de versionado de archivos, por etiqueta, fechas o ramas.

5.3. Sistemas de control de versiones

Las funciones principales de los sistemas de control de versiones son proporcionar un histórico de los cambios de los archivos de un proyecto y permitir recuperar una determinada versión del archivo en cualquier momento. Imaginemos que tenemos la versión 1.00 de un archivo de instrucciones de instalación de un programa. A este archivo de texto le añadimos varios párrafos que indican los pasos detallados que hay que seguir para instalar nuestro programa en versiones de UNIX BSD y esta nueva versión con instrucciones para BSD se convierte en la versión 1.01 del documento. A medida que el proyecto sigue avanzando, este archivo continúa evolucionando. Con un sistema de control de versiones, en cualquier momento podemos obtener la

ANOTACIONES

versión 1.00 o 1.01 sin necesidad de mantener copias separadas de los archivos.

Los sistemas de control de versiones se basan en mantener todos los archivos del proyecto en un lugar centralizado, normalmente un único servidor. Aunque también hay sistemas distribuidos, donde los desarrolladores se conectan y descargan una copia en local del proyecto. Con ella, envían periódicamente los cambios que realizan al servidor y van actualizando su directorio de trabajo que otros usuarios a su vez han ido modificando.

Los sistemas de control de versiones de código están integrados en el proceso de desarrollo de software de muchas empresas. Cualquier empresa que tenga más de un programador trabajando en un mismo proyecto acostumbra a tener un sistema de este tipo, y a medida que crece el número de personas que se involucran en un proyecto, más indispensable se hace un sistema de control de versiones.

5.3.1. Algunos términos comunes

Los sistemas de control de versiones tienen una terminología muy específica para referirse a conceptos singulares de este tipo de sistemas. A continuación, veremos los conceptos y términos más habituales del paquete de software CVS y que son extensibles a la mayoría de sistemas:

Acceso anónimo. Tipo de acceso a un servidor con el cual tenemos derecho a ver y copiar los archivos del repositorio, pero no a modificarlos.

Repositorio. Lugar centralizado donde se almacena la copia de todos los archivos del proyecto que compartimos con todos los usuarios y que habitualmente reside en un único servidor.

Directorio de trabajo. Versión del proyecto que proviene del repositorio y que un usuario tiene en su disco duro local y sobre la cual trabaja.

Commit. Acción con la que enviamos los cambios que hemos realizado en nuestro directorio de trabajo al repositorio central del servidor.

ANOTACIONES

También se utiliza para añadir de forma definitiva nuevos archivos al repositorio.

Rama. A partir de un punto concreto se crea una bifurcación del proyecto que puede evolucionar por separado.

Etiqueta. Nombre simbólico que asignamos a un archivo o grupo de archivos del proyecto para indicar que la versión tiene un significado especial en nuestro proceso de desarrollo. Con este nombre nos podremos referir a los archivos más adelante.

Tarball. Copia completa de un proyecto en un punto determinado realizada con el comando tar de Unix. Es muy frecuente realizar una copia de todos los archivos de un proyecto en este formato porque es mucho más rápido de descargar por Internet que mediante CVS o Subversion.

5.3.2. Características de los sistemas de control de versiones

Todos los sistemas de control de versiones modernas permiten un conjunto de funciones que se consideran indispensables para conocer exactamente el estado de un proyecto. A continuación enumeramos las funciones principales:

- **Control de usuarios.** Establecer qué usuarios tienen acceso a los archivos del proyecto y qué tipo de acceso tienen asignado.
- **Mantener un control detallado de cada archivo.** Disponer de un control de cambios que se han efectuado en el archivo, en qué fecha, con qué motivo, quién los realizó y mantener las diferentes versiones.
- **Mezclar dos versiones diferentes del mismo archivo.** En caso de que dos usuarios modifiquen un archivo (sistemas de no bloqueo de archivos) proporcionar herramientas para gestionar este tipo de conflictos.
- **Bifurcación de proyectos.** A partir de un punto concreto, crear dos ramas del proyecto que pueden evolucionar por separado. Es

ANOTACIONES

habitual utilizar esta técnica cuando hemos liberado una versión de un producto y necesitaremos que siga evolucionando.

5.3.3. Principales sistemas de control de versiones

Existe una buena oferta de sistemas de control de versiones disponibles como software propietario y libre, aunque durante este capítulo nos centraremos en los sistemas CVS y Subversion que son soluciones libres. Mencionaremos brevemente los principales sistemas de control de versiones existentes en el mercado:

- CVS (concurrent versions system)

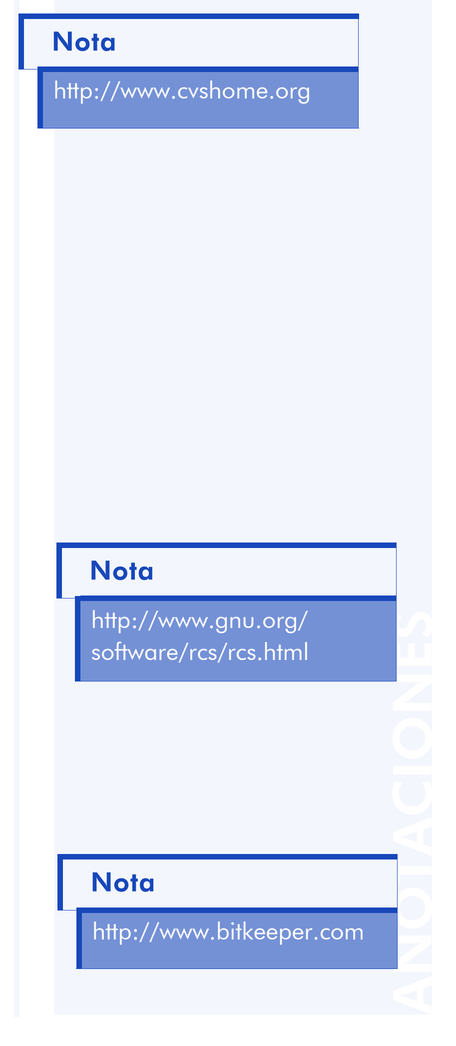
Es el sistema más utilizado en el mundo del software libre para el control de versiones. Basado en el modelo cliente-servidor, el propio sistema es software libre y existen versiones del software para multitud de plataformas. Su solidez y su probada eficacia, tanto en grupos pequeños de usuarios como en grandes, le han convertido en la herramienta que utilizan proyectos de software libre como Mozilla, OpenOffice.org, KDE o GNOME, por mencionar solamente algunos.

- RCS (*revision control system*)

El sistema CVS en realidad está basado en otro anterior RCS (*revision control system*) que fue desarrollado por Walter Tichy en la Universidad de Purdue a principios de 1980. El sistema RCS es extremadamente simple y fácil de instalar, pero no puede cubrir las necesidades de proyectos o equipos medianos. Entre las limitaciones más notables de RCS, destaca que sólo puede trabajar en un único directorio cuando la mayoría de proyectos tienen múltiples directorios. RCS utiliza un sistema de bloqueo de archivos que impide a varios desarrolladores trabajar sobre el mismo archivo.

- BitKeeper

BitKeeper es un producto propietario desarrollado por la empresa BitMover. Es probablemente el producto más sofisticado de su categoría. Entre las características que lo diferencian del resto de productos, destacan la posibilidad de trabajar con repositorios distribuidos



y un sistema muy avanzando para integrar diferentes versiones de un mismo archivo.

- Microsoft Visual Source Safe

Nota

http://msdn.microsoft.com/ssafe

Es uno de los productos más utilizados para desarrollo de aplicaciones Windows. Principalmente porque se integra en el entorno de trabajo de Visual Studio y con el resto de herramientas de desarrollo de Microsoft. Tiene funciones de comparación visual de archivos realmente avanzadas, en su modo básico de funcionamiento utiliza bloqueo de archivos.

5.3.4. Sistemas de compartición

Un punto crítico para los sistemas de control es la gestión de un archivo que es utilizado por varios usuarios al mismo tiempo. El mecanismo que utilicen para resolver este tipo de situación tendrá implicaciones en la forma como los usuarios pueden trabajar simultáneamente con el archivo y como se gestionan los cambios que se realizan. Existen dos sistemas de compartición:

- Por bloqueo de archivo

Es el sistema más primitivo y el que utilizan RCS y las versiones antiguas de *Visual Source Safe de Microsoft*. Se trata de bloquear el archivo cuando un usuario está haciendo cambios de tal modo que los otros usuarios no pueden realizar modificaciones hasta que no finalice el usuario.

Es habitual que un usuario empiece a trabajar en un archivo y necesite varias horas para añadir el nuevo código o modificar el existente, por esta razón, estos sistemas no son muy adecuados, ya que un mismo usuario puede bloquear un archivo durante un periodo largo de tiempo durante el cual ningún otro usuario puede realizar cambios.

- Por gestión de cambios

Este sistema permite que cualquier usuario o grupos de usuarios modifiquen al mismo tiempo un archivo. Cuando han finalizado los

cambios, realizan una operación *commit*, que comprueba las diferencias entre el archivo en el servidor y el mejorado por el desarrollador y envía los cambios. Éste es el método que utiliza CVS.

El método es una mejora substancial sobre el sistema de bloqueo de archivos, ya que permite a más de un usuario trabajar sobre el mismo archivo, pero introduce un nuevo problema, la gestión de conflictos. Imaginemos que tenemos dos personas que están trabajando sobre el mismo archivo y sobre la misma zona del archivo, cuando el primer usuario envíe sus cambios, éstos se guardarán en el servidor, y poco después, cuando el segundo usuario envíe los suyos, se encontrará que la copia que tenía en local ha cambiado y que los cambios no se pueden integrar automáticamente. Disponemos de sistemas sofisticados para gestionar lo mejor posible este tipo de situaciones, incluyendo herramientas visuales. CVS utiliza un sistema rudimentario, donde básicamente nosotros mismos habremos de reimplementar los cambios.

5.4. Primeros pasos con CVS

5.4.1. Instalación de CVS

A través del sitio web www.cvshome.org, se pueden obtener las diferentes versiones cliente y servidor del programa CVS. La mayoría de distribuciones GNU/Linux incluyen paquetes preparados con las herramientas CVS. De hecho, en la mayoría vienen instalados por defecto.

Si somos usuarios de Debian y no tenemos instalado CVS, su instalación, como la del resto de software de esta distribución, es muy sencilla:

```
$ apt-get install cvs
```

Para usuarios de la distribución Red Hat, lo mejor es recurrir a los CD-ROM originales y realizar la instalación desde el entorno gráfico o con la siguiente orden de línea de comandos:

```
$ rpm -ivh cvs
```

ANOTACIONES

Durante este temario nos referiremos exclusivamente a las versiones GNU/Linux de las herramientas CVS. Si necesitamos trabajar en Microsoft Windows existen al menos dos clientes:

- Cygwin

Cygwin es una emulación completa de un entorno tipo Unix para Windows. Disponemos de la mayoría de herramientas que existen en GNU/Linux, incluyendo una versión adaptada de CVS. De hecho, éste es el entorno en que se desarrollan algunos proyectos libres, como Abiword en su versión Windows.

La ventaja de Cygwin es que estamos trabajando con la versión adaptada de la herramienta original de CVS para Unix en entorno Windows, de tal manera que su comportamiento es idéntico a la versión GNU/Linux descrita en este módulo.

Nota

http://www.cygwin.com/
http://www.abiword.com

- WinCVS

WinCVS es un programa libre con entorno gráfico para plataforma Windows que permite trabajar con repositorios CVS. Es altamente intuitivo y no necesita conocer los comandos CVS, ya que desde el entorno gráfico se pueden realizar todas las operaciones necesarias.

Nota

http://www.wincvs.org

5.4.2. Obtener un directorio de trabajo de un proyecto ya existente

Una buena forma de tener un primer contacto con CVS es utilizar un servidor de un proyecto de software libre y obtener una copia de los archivos del proyecto.

El sitio web www.mozilla.org alberga el proyecto Mozilla. Este proyecto tiene como objetivo producir las tecnologías necesarias que permitan construir un conjunto de herramientas para trabajar en Internet. Los componentes principales de Mozilla son un navegador web, un sistema de correo electrónico y un editor de páginas web,

Nota

En la página http://mozilla.org/cvs.html hay información detallada sobre cómo acceder a este servidor mediante CVS.

ANOTACIONES

aunque el número de herramientas que se van introduciendo crece constantemente.

Primero definiremos la variable de entorno CVSROOT, que indica el servidor, protocolo de acceso, y usuario que utilizaremos para acceder al repositorio CVS. En el caso del proyecto Mozilla, definiremos la variable de entorno con el siguiente valor:

```
$ CVSROOT=:pserver:anonymous@cvs-mirror.mozilla.org:/cvsroot
$ export CVSROOT
```

Si prestamos atención a la línea, observaremos que nuestro nombre de usuario es *anonymous* y ya forma parte de la cadena CVSROOT, de este modo sólo tenemos que introducir la contraseña para identificarnos con acceso anónimo. También es posible usar el parámetro -d del comando CVS para especificar el camino al servidor CVS.

Ahora podemos teclear el siguiente comando:

```
$ cvs login
```

CVS contesta:

```
Logging in to :pserver:anonymous@cvs-mirror.mozilla.org:2401/cvsroot
CVS password:
```

Este comando es el que nos permite autentificarnos en el servidor. La contraseña del usuario *anonymous* para el proyecto Mozilla es *anonymous*. En algunos servidores para el acceso anónimo la contraseña es simplemente <retorno>, es decir, sin contraseña. En cualquier caso, depende de cómo haya configurado el administrador el sistema repositorio.

A continuación podemos utilizar el comando *checkout* para obtener el código del proyecto Mozilla. Este comando es el que utilizamos la primera vez que descargamos un proyecto de un repositorio, y admite como parámetro el nombre del módulo. Para el proyecto Mozilla teclearemos:

```
$ cvs checkout mozilla
```

ANOTACIONES

ANOTACIONES

CVS muestra los nombres de los archivos que se van descargando:

```
cvs server: Updating mozilla
U mozilla/.cvsignore
U mozilla/LEGAL
U mozilla/LICENSE
U mozilla/Makefile.in
U mozilla/README.txt
U mozilla/aclocal.m4
...
```

El nombre de los módulos, en este caso Mozilla, es arbitrario y definido por el administrador del repositorio. De hecho, los nombres de los módulos son los nombres de los directorios raíz del proyecto.

Si examinamos la estructura de directorios creada, observaremos que existe un directorio CVS para cada subdirectorio del proyecto. Estos directorios contienen tres archivos:

- **Entries**. Contiene los archivos de este directorio que están reflejados en el repositorio.
- **Repository**. Contiene la ruta relativa al módulo al que pertenece este archivo.
- **Root**. Contiene la ruta principal al servidor CVS.

Bajo ningún concepto deben borrarse, y en caso de editarse, debe hacerse con extremo cuidado.

> ***Nota***
>
> Podemos practicar con otros proyectos libres, encontraremos instrucciones detalladas de cómo descargar el proyecto GNOME vía CVS en el sitio web www.gnome.org.

5.4.3. Sincronizándose con el repositorio

Una vez disponemos de una copia en local de los archivos, ya tenemos una copia del proyecto con la que podemos empezar a trabajar localmente. Sin embargo, a medida que pasan los días, otros usuarios

van efectuando cambios en el repositorio del proyecto y en consecuencia nuestro directorio de trabajo queda desactualizado debido a que estos nuevos cambios no se han aplicado. Es habitual que nos sincronicemos con el repositorio regularmente para mantener nuestro directorio de trabajo lo más actualizado posible del proyecto. Esta operación se realiza con el comando *update*, de la siguiente manera:

```
$ cvs update -d -P
```

El comando *update* acepta diversos parámetros, pero los parámetros -d y -P son muy habituales, el primero construye los directorios necesarios y el segundo elimina de nuestro directorio de trabajo los directorios que han quedado vacíos.

El comando *update*, cuando informa de los archivos actualizados, añade una letra al principio del nombre de archivo. Estas letras tienen el siguiente significado:

- C, indica que hay un conflicto en el archivo. Ha habido una actualización en el repositorio y en el directorio de trabajo y debe solucionarse manualmente el conflicto.

- U, el directorio de trabajo que teníamos era antiguo y se ha actualizado con la versión más reciente del repositorio.

- M, el directorio de trabajo del archivo tiene cambios pero se ha conseguido una actualización automática con la versión más reciente del repositorio.

- A, si el archivo ha sido añadido pero aún no hemos efectuado el comando *commit*.

- R, si el archivo ha sido borrado pero aún no hemos efectuado el comando *commit*.

Durante este proceso de sincronización, sólo los archivos que han sido modificados son actualizados. Es importante entender que el comando *update* en ningún caso envía las modificaciones que hayamos realizado en nuestro directorio de trabajo de los archivos al repositorio (subir

ANOTACIONES

cambios al repositorio), sino que simplemente nos actualiza nuestro directorio de trabajo con los cambios del repositorio (bajar los cambios y actualizar el directorio de trabajo).

5.4.4. Cambios al repositorio

Durante el proceso normal de trabajo en un proyecto, es habitual ir modificando archivos a medida que vamos introduciendo mejoras. Una vez hemos finalizado los cambios, éstos deben enviarse al repositorio para que todos los usuarios puedan acceder a las mejoras que hemos efectuado.

Antes de enviar los cambios, resulta interesante ver exactamente qué cambios hemos realizado. Para ello tenemos el comando *diff* que sigue la siguiente sintaxis:

```
$cvs diff -u nombre_archivo
```

Este comando nos muestra las diferencias de nuestro directorio de trabajo respecto al repositorio, de esta manera podemos ver qué cambios exactamente hemos ido realizando. Hasta ahora hemos visto cómo determinar las diferencias. Ahora veremos cómo enviar estas diferencias al repositorio, el comando *commit* es el encargado de enviar los cambios.

Por ejemplo, modifiquemos el archivo 'README.TXT' del proyecto Mozilla que hemos descargado previamente y que se encuentra en el directorio principal. A continuación introduzcamos:

```
$cvs commit
```

Después de procesar todos los archivos del proyecto en busca de archivos con modificaciones, mostrará una lista de los cambios realizados; por defecto esta lista se muestra en el editor vi.

Podemos definir la variable de entorno CVSEDITOR para cambiar el editor por defecto que deseamos usar. Como respuesta al comando *commit* CVS mostrará:

ANOTACIONES

```
CVS: ----------------------------------------------------------------------------------------
CVS: Enter Log. Lines beginning with `CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Modified Files:
CVS: README.txt
CVS: ----------------------------------------------------------------------------------------
```

Si deseamos continuar, teclearemos: `!q`, que es la orden del editor vi para salir sin guardar los cambios del registro que se nos acaba de mostrar. A continuación nos preguntará si deseamos enviar los cambios al servidor. CVS nos responderá:

```
cvs [server aborted]: "commit" requires write access to the repository
```

Que es normal en este caso, ya que como usuario anónimo del proyecto Mozilla no tenemos derecho a modificar los archivos. El comando *commit* tiene el parámetro -m (mensaje) que es muy utilizado para indicar comentarios, y en algunos proyectos es casi de uso obligado. Por ejemplo:

```
cvs commit -m "Solventa el problema con el script de instalación"
```

Los cambios de los archivos modificados serán enviados al repositorio y registrados con el comentario indicado. Los comentarios resultan fundamentales durante el proyecto para entender por qué nosotros u otras personas han introducido los cambios en el proyecto.

5.4.5. Publicando cambios con *diff y patch*

No es extraño que en un proyecto del que no somos colaboradores habituales encontremos un error en el programa, incluso es probable que tengamos la solución y queramos enviarla a los responsables del proyecto. Como es lógico, no tendremos acceso de escritura al repositorio de CVS del proyecto, de forma que no podremos publicar la solución al repositorio directamente. Sin embargo, sí que podemos publicar nuestra solución al problema en una lista de correo o

ANOTACIONES

enviarlo por correo electrónico al responsable del proyecto. Para ésta y otras situaciones existen la pareja de comandos *diff* y *patch*.

Imaginemos que hemos modificado un archivo para corregir un error. Para crear un archivo con las diferencias que hemos introducido, haremos:

```
$diff -c nombre_primer_archivo nombre_segundo_archivo > fix.patc
```

Por defecto, el programa *diff* envía la salida a pantalla, por esto redirigiremos la salida al archivo fix.patch. Este archivo contendrá los cambios.

La persona que reciba el archivo fix.patch podrá recuperar las modificaciones que hemos realizado utilizando el comando *patch*, de la siguiente forma:

```
$patch < archivo_para_aplicar_parches < parche
```

Como hemos podido observar, ésta es una forma muy sencilla y eficiente de enviar correcciones a otros usuarios y es una práctica muy común en proyectos de software libre.

5.5. Creación y administración de repositorios

La creación de un repositorio es una tarea relativamente sencilla. Primero debemos escoger en qué directorio del sistema de archivos vamos a ubicar el repositorio. Un buen lugar puede ser /var/lib/cvs. Es importante tener acceso de escritura al directorio para que, cuando ejecutemos CVS, pueda crear los archivos necesarios.

Una vez tenemos los permisos, creamos el directorio donde se ubicará el repositorio:

```
$mkdir cvs
```

Tendremos el directorio /var/lib/cvs. Como veremos más adelante, CVS utiliza la gestión de usuarios y permisos del sistema operativo.

ANOTACIONES

Los comandos CVS son ejecutados con el identificador del usuario que ejecuta el comando. Es muy importante asegurarnos de que todos los usuarios tendrán acceso al repositorio. Un método sencillo es asignar al directorio el grupo *users* para que todos los usuarios que pertenecen a este grupo puedan acceder al repositorio:

```
$chgrp users /var/lib/cvs
```

Luego debemos asignar a este grupo todos los usuarios que deseamos que tengan acceso al repositorio.

Ahora ya tenemos todo preparado para que CVS inicialice el repositorio y cree los archivos necesarios para su gestión:

```
$ cvs -d /var/lib/cvs init
```

El comando *init* creará el repositorio en el directorio central e inicializará los archivos necesarios. El parámetro -d es necesario, ya que indica el directorio donde se ubicará el repositorio. El comando no retorna ningún resultado pero podremos observar cómo ha creado un directorio en /var/lib/cvs/CVSROOT que contiene todos los archivos que necesita CVS para gestionar el repositorio.

Para poder compartir el repositorio con otros usuarios, nos interesa ejecutar CVS en modo servidor. CVS utiliza el puerto TCP 2401. Para instalar este servicio, seguiremos los siguientes pasos:

1) Editamos el archivo de configuración de CVS, llamado cvs.conf. Este archivo se encuentra en el directorio /var/lib/cvs. Hemos de añadir la siguiente línea que indicará dónde se halla el repositorio:

```
CVS_REPOS="/var/lib/cvs"
```

2) Necesitamos activar el servicio. Para ello deberemos crear un archivo llamado cvspserver en el directorio /etc/xinetd.d. Este archivo indica la descripción del servicio.

ANOTACIONES

Nota

Es importante comprobar qué CVS se encuentra realmente instalado en /usr/bin/cvs

```
service cvspserver
{
 port = 2401
 disable = no
 socket_type = stream
 protocol = tcp
 wait = no
 user = root
 server = /usr/local/bin/cvs
 server_args = -f --allow-roof = /var/lib/cvs pserver
 }
```

Una vez activado el demonio y creado el repositorio, nos podemos conectar como hemos hecho hasta ahora. Algunas distribuciones como Debian usan el demonio inetd y la forma de configuración difiere a la comentada.

```
$ CVSROOT=:pserver:anonymous@NOMBRE_DE_NUESTROPC:/var/lib/cvs
$ export CVSROOT
$ cvs login
```

En la línea CVSROOT, debemos substituir NOMBRE_DE_NUESTROPC por el nombre de nuestra máquina GNU/Linux.

Para identificarnos, cualquier nombre de usuario –con los privilegios adecuados y con su correspondiente contraseña– que tengamos definido en el sistema donde hemos instalado el servicio será válido para acceder al repositorio.

5.5.1. Tipos de conexión al repositorio

Hemos visto la conexión al repositorio mediante el método *pserver*. Este sistema es muy básico y se basa en enviar las contraseñas sin encriptar, de forma muy similar a como lo hacen los protocolos asociados a las aplicaciones Telnet o FTP. Este nivel de seguridad no es suficiente para proyectos donde garantizar la seguridad del repositorio sea un factor importante. También debemos recordar que los usuarios de CVS son válidos en la propia máquina, por lo que un problema de seguridad con CVS se extiende a toda la máquina.

CVS proporciona la posibilidad de trabajar con métodos externos de autentificación para mejorar la seguridad. El sistema más extendido

ANOTACIONES

es SSH (*secure shell*), que permite conexiones cifradas, la contraseña viaja cifrada por la red, y además permite asegurarnos de que el huésped al que nos conectamos es el correcto. Los repositorios de SourceGear por ejemplo sólo aceptan conexiones con un método seguro, en concreto SSH.

Para activar la conexión mediante SSH, es necesario definir la variable de entorno CVS_SSH de la siguiente manera:

```
$ CVS_SSH=ssh
$ export CVS_SSH
```

El método de conexión como hemos visto se define en la variable CVSROOT. La utilización de SSH requiere la correcta instalación del software OpenSSH.

5.5.2. Importando proyectos ya existentes

Una vez el repositorio está inicializado, podemos crear la estructura de directorios y un proyecto nuevo, o bien importar un proyecto ya existente. En cualquier caso, es importante que decidamos previamente la estructura de directorios que queremos que tenga nuestro proyecto, ya que mover archivos con CVS no es muy cómodo.

El comando *import* de CVS nos facilita la importación de proyectos ya existentes al repositorio. Con *import*, podemos importar un archivo o conjunto de archivos como parte de un proyecto. Por ejemplo, importemos todo el directorio /home/jordi

```
$cvs import directorio vendortag releasetag
```

En nuestro caso:

```
$cvs import codigo vendorta releasetag
```

Donde CVS responde:

```
I codigo/simul.c
I codigo/comun.c
I codigo/corto/etapas.c
```

Nota

www.sourcegear.net

ANOTACIONES

Dependiendo del número y nombre de archivos que contenga el directorio.

Los parámetros *vendortag* y *releasetag* se utilizan para el control de versiones, en nuestro caso utilizamos dos nombres arbitrarios.

5.5.3. Añadiendo archivos o directorios

Para añadir archivos en un repertorio CVS, disponemos del comando *add*. Este comando permite tanto añadir un archivo o grupo de archivos, como añadir un directorio completo. Para añadir un archivo tecleamos:

```
$ cvs add nombre_de_archivo
```

CVS responde:

```
cvs server: scheduling file `nombre_de_archivo' for addition
cvs server: use 'cvs commit' to add this file permanently
```

Los archivos no se añadirán de forma definitiva al repositorio hasta que realicemos una orden *commit*, para enviar todos los cambios al servidor, es decir, hasta que ejecutemos:

```
$ cvs commit
```

que enviará los cambios al servidor y los hará efectivos.

5.5.4. Los archivos binarios

La mayoría de herramientas relacionadas con CVS que hemos visto funcionan sobre archivos de texto, ya que de hecho es para lo que CVS fue originalmente diseñado. Sin embargo, en un proyecto cualquiera es habitual encontrar archivos que no son propiamente de texto, a los que llamamos binarios. Entre los archivos binarios más habituales, tenemos los archivos gráficos, de sonido, o versiones de un software compiladas que mantenemos en el CVS.

ANOTACIONES

El problema principal radica en que CVS utiliza dos características, la expansión de palabras clave y la conversión de retornos de carro que, aplicadas a archivos binarios, modificarán su contenido haciéndolos inservibles.

Si añadimos un archivo binario, es importante desactivar estas funciones para este archivo. Para añadir archivos binarios, debemos hacer:

```
$cvs add -kb nombre_de_archivo
```

El parámetro -kb desactiva las características que pueden dañar un archivo binario. Como en el caso de los archivos de textos, los cambios no se enviarán hasta que ejecutemos el comando *commit*.

5.5.5. Eliminando archivos y directorios

La eliminación de archivos del repositorio es una operación que debemos efectuar con cuidado. Para poder eliminar un archivo, primero lo debemos eliminar de nuestro directorio de trabajo, es decir, de nuestro disco, y después del repositorio. La secuencia de instrucciones es la siguiente:

```
$ rm nombre_de_archivo
$ cvs remove nombre_de_archivo
```

Como en el caso de añadir archivos, la operación no se completa hasta que ejecutamos el comando CVS *commit*.

El método visto hasta ahora sirve para eliminar archivos, pero no para eliminar directorios. De hecho, CVS no permite eliminar directorios debido a cómo organiza internamente los repositorios. Lo único que podemos hacer es eliminar todos los archivos del directorio, dejarlo vacío, y utilizar siempre el parámetro -P del comando *update* cuando nos sincronicemos con el repositorio, para que no se creen en nuestro directorio de trabajo los directorios vacíos.

ANOTACIONES

5.5.6. Moviendo archivos

CVS no dispone de ningún comando específico para poder mover archivos de un lugar a otro del repositorio. Sin embargo, podemos emular esta funcionalidad combinando varias operaciones de CVS.

Para mover un archivo, lo copiaremos de su lugar de origen en nuestro directorio de trabajo al lugar de destino, es decir, el lugar donde queremos que resida en el futuro. Una vez realizada esta operación sobre el repositorio, eliminaremos el archivo del lugar de origen y añadiremos el nuevo archivo.

La secuencia de comandos es la siguiente:

```
$mv dirorigen/origen dirdestino/destino
$cvs add dirdestino/destino
$cvs remove dirorigen/origen
```

Los cambios no se harán efectivos hasta que se ejecute el comando commit. Es importante destacar que con esta operación perderemos el histórico de etiquetas y comentarios realizados en el repositorio, ya que efectivamente estamos incluyendo un archivo nuevo.

5.6. Trabajando con versiones, etiquetas y ramas

5.6.1. Etiquetas y revisiones

El desarrollo de un proyecto es un proceso vivo donde constantemente van produciéndose cambios como parte del proceso continuo de mejora. En algunos puntos del proyecto interesa tener la capacidad de poder indicar que en este preciso momento los archivos que hay en el repositorio forman una determinada versión. Este proceso se llama etiqueta. Es muy utilizado para marcar una versión exacta que liberamos para después podernos referir de forma sencilla.

Podemos establecer una etiqueta en el repositorio fácilmente:

```
$ cvs tag Version100
```

ANOTACIONES

De esta forma etiquetaremos todos los archivos del proyecto con la etiqueta "Version100". A medida que el proyecto sigue evolucionando, puede ser necesario recuperar una versión que habíamos marcado anteriormente. Para ello, utilizaremos el comando *update* que ya conocemos:

```
$ cvs update -r Version100
```

De esta manera obtendremos una nueva copia del proyecto tal como estaba en el momento de marcarlo como "Version100". Sin embargo, esto representa un pequeño inconveniente, ya que el directorio de trabajo retrocederá completo al estado en que se encontraba cuando efectuamos la etiqueta, probablemente hace unos meses. Acostumbra a ser mucho más práctico descargar una versión determinada en un directorio aparte, podemos hacerlo de la siguiente manera:

```
$ cvs checkout -r Version100
```

También es posible realizar las mismas operaciones basadas en la fecha del archivo. Resulta más intuitivo trabajar con etiquetas, ya que son nombres que pueden tener un significado y son más fáciles de asociar que las fechas.

5.6.2. Creación de ramas

La creación de ramas es muy útil cuando trabajamos en paralelo con varias versiones del proyecto. Por ejemplo, si vamos a liberar una versión 1.0 de un producto. En ese momento, creamos dos ramas del proyecto, una donde está la versión 1.0 y donde podemos seguir corrigiendo los posibles errores que aparezcan y más adelante crear una versión 1.01. Al mismo tiempo, podemos tener otra rama en paralelo donde el proyecto ya ha evolucionado a la versión 1.1 y aún se encuentra en fase de desarrollo. Las ramas permiten trabajar con varias versiones del mismo proyecto de forma concurrente.

Podemos crear una rama con el siguiente comando:

```
cvs tag -b nombre_de_la_rama archivo
```

ANOTACIONES

A partir de ese momento podemos continuar operando con la rama principal de CVS y cuando nos queramos referir a la rama anteriormente creada, podremos hacerlo usando el nombre_de_la_rama escogido.

5.7. Bonsai: gestión de CVS vía web

Nota

En http://www.mozilla.org/bonsai.html podéis encontrar la página principal del proyecto Bonsai, con su código fuente e instrucciones detalladas para su instalación.

Bonsai es una herramienta que fue creada en el entorno del proyecto de software libre Mozilla. Esta herramienta permite la gestión vía web de un repositorio CVS. Su utilidad radica en que añade una interfaz visual y es extremadamente útil para hacer seguimientos de proyectos donde hay cambios constantes en el repositorio.

En este capítulo, nos limitaremos a ver su funcionalidad desde la perspectiva del usuario.

Entre las características que ofrece Bonsái, destacan:

- Posibilidad de ver los últimos cambios en el repositorio, quién los ha efectuado, a qué hora, el comentario del cambio, etc.
- Ver qué cambios se han efectuado por un usuario determinado, en una rama del proyecto, o en un periodo de tiempo.
- Ver un determinado archivo con anotaciones de quién fue la última persona que cambió cada línea del archivo.
- Comparación visual de los diferentes cambios que se han producido en el repositorio.

Bonsai sólo permite hacer seguimiento del proyecto, no permite hacer cambios en el mismo. A través de la página http://bonsai.mozilla.org/, podemos efectuar todas las operaciones descritas sobre el repositorio del proyecto libre Mozilla.

Tabla de comandos de CVS

La tabla que viene a continuación contiene una descripción de los principales comandos de CVS.

Tabla 1. Descripción de los principales comandos de CVS

Comando	Descripción
add	Añade un nuevo archivo o un conjunto de archivos de un directorio al repositorio.
admin	Interfaz de administración.
annotate	Muestra el archivo indicando el número de revisión de cada línea del archivo y qué usuario fue el último que la modificó.
checkout	Obtiene una copia de los archivos del repositorio y crea nuestro directorio de trabajo.
commit	Acción con la que enviamos los cambios que hemos realizado en nuestro directorio de trabajo al repositorio central del servidor.
diff	Muestra las diferencias de un archivo o grupo de archivos entre nuestro directorio de trabajo al repositorio central del servidor.
edit	Edita los archivos bajo observación.
editors	Muestra quién está editando archivos bajo observación.
export	Exporta archivos de CVS, similar a *checkout*.
history	Muestra el histórico de acceso de usuarios al repositorio CVS.
import	Importa archivos al repositorio CVS.
init	Crea e inicializa un nuevo repositorio.
log	Muestra el historial de los archivos.
login	Autentifica el usuario. Con este comando introducimos la contraseña de nuestro usuario y nos valida en el servidor.
logout	Desconecta al usuario.
pserver	Servidor en modo contraseña.
rannotate	Muestra el archivo indicando quién ha modificado cada línea del archivo.
rdiff	Crea un parche con la diferencia entre dos versiones de un archivo.
release	Indica que el módulo ya no está en uso.
remove	Elimina un objeto del repositorio.
rlog	Muestra el histórico para un módulo del repositorio.
rtag	Añade una etiqueta.
server	Modo servidor.
status	Muestra la información de estado de los archivos.
tag	Añade una etiqueta a los archivos del repositorio.
unedit	Deshacer una modificación realizada por un comando de edición.
update	Actualiza el directorio local de trabajo con la copia en el servidor.
version	Muestra la versión actual del programa CVS.
watch	Establece los archivos bajo observación.
watchers	Muestra los usuarios que están observando archivos.

ANOTACIONES

5.7.1. Subversion

Subversion es un software libre de control de versiones que nació en el año 2001 con el objetivo de solventar las principales carencias de CVS. Muchos desarrolladores ven en Subversion el sistema de control de versiones que sustituirá a CVS.

Entre las principales mejoras de Subversion respecto a CVS, destacan:

- Posibilidad de mover archivos. Una de las grandes limitaciones de CVS ha sido la carencia de un comando para poder mover archivos entre diferentes partes del repositorio. Subversion finalmente corrige esta carencia.

- *Commits* atómicos. La posibilidad de enviar un conjunto de cambios al repositorio en bloque evitando los problemas que suceden en CVS cuando enviamos una gran cantidad de cambios al repositorio que son todos parte de un mismo trabajo, pero sólo parte de ellos son aceptados por CVS.

- Metadatos. Subversion permite guardar por cada archivo o directorio un conjunto de llaves emparejadas con su valor que nos permiten almacenar metadatos y luego recuperarlos.

- Versionado por directorios. CVS sólo ofrece control de versiones por archivo, con Subversion podemos tener también control por directorio.

- Soporte para diferentes transportes de red. Subversion ha sido diseñado para que sea muy sencillo añadir nuevos transportes de red, como conexiones seguras mediante SSH o mediante WebDAV.

- La comparación de archivos se realiza con un algoritmo binario que es mucho más efectivo.

5.7.2. Instalación de Subversion

A través del sitio web http://subversion.tigris.org/ se pueden obtener las diferentes versiones cliente y servidor del programa Subversión,

ANOTACIONES

así como su código fuente. La mayoría de distribuciones GNU/Linux incluyen paquetes preparados con las herramientas Subversion. Podemos descargar de este sitio web la versión especifica de nuestra distribución GNU/Linux o bien descargar el código fuente, compilarlo e instalarlo nosotros mismos como haríamos con cualquier otro paquete de software.

Subversion instala varias herramientas, pero las principales son:

- **svn**. El cliente de Subversion. Con él realizaremos las operaciones principales como obtener una copia de trabajo, sincronizar nuestra copia con el servidor o enviar los cambios realizados.

- **svnadmin**. La herramienta de creación y administración de repositorios. Ésta es la herramienta principal de administración de Subversion.

- **svnserve**. Servidor ligero que funciona como demonio, usualmente en el puerto 3690, que permite a clientes svn conectarse mediante conexiones TCP/IP a través de los protocolos svn:// y svn+ssh://.

- **svnlook**. Herramienta de administración de repositorios que nos permite examinar el histórico del repositorio.

Existen también algunos clientes gráficos para diferentes plataformas:

- **TortoiseSVN**

Es un cliente gráfico de Subversion para Microsoft Windows implementado como una extensión de la shell de Windows.

Nota

http://tortoisesvn.tigris.org/download.html

- **Rapid SVN**

Rapid SVN es un cliente SVN multiplataforma que funciona sobre GNU/Linux, Mac y Windows y que proporciona un entorno sencillo para trabajar con Subversion.

Nota

http://rapidsvn.tigris.org/

ANOTACIONES

5.7.3. Obtener un directorio de trabajo de un proyecto ya existente

Para poder trabajar con un proyecto que utiliza Subversión, necesitamos empezar obteniendo un directorio de trabajo de los archivos del proyecto. Al igual que con CVS, usamos el comando *checkout* para este propósito. Podemos hacerlo de la siguiente manera:

```
$ svn checkout repositorio
```

Donde *repositorio* es el repositorio y módulo del que obtendremos una copia, Subversion utiliza URI para especificar las rutas de los repositorios. Esto permite utilizar una forma estándar para describir localizaciones a las cuales podemos acceder bajo diferentes transportes, como file:/// (archivo local), http:// (vía web) o https:// (vía web con conexión segura).

Nota

http://www.mono-project.com

Por ejemplo, para obtener una copia del repositorio Subversion del proyecto Mono, usaremos:

```
$ svn checkout svn+ssh://user@mono-cvs.ximian.com/source/trunk/mono
```

Donde svn+ssh es el tipo de transporte, user el usuario que utilizaremos, y el resto la ruta a la máquina en la red que contiene el repositorio.

5.7.4. Creación de repositorios

Una vez instalado Subversión, debemos crear un repositorio donde se almacenarán nuestros archivos. Para ello, primero nos situaremos en el directorio donde queremos crear el repositorio y luego usaremos la herramienta de administración svnadmin para crearlo con los siguientes comandos:

```
$mkdir repositorio
$svnadmin create repositorio
```

ANOTACIONES

Nota

Para el resto de los ejemplos asumimos que el directorio repositorio se ha creado en el subdirectorio /home/jordi/subversión, por lo que el directorio completo al repositorio es /home/jordi/subversion/repositorio.

El segundo comando crea todos los archivos necesarios para el control del repositorio en el directorio llamado repositorio, que es donde reside el repositorio que acabamos de crear.

A continuación, lo más usual es importar a nuestro repositorio un proyecto que ya tengamos.

El comando *import* nos facilita esta tarea. Con *import*, podemos importar un archivo o conjunto de archivos como parte de un proyecto. Por ejemplo, importemos todo el directorio /home/jordi en el repositorio que hemos creado. Para ellos tecleamos:

```
$svn import /home/jordi/ file:///home/jordi/subversion/repositorio
```

El comando mostrará todos los archivos que se han importado al repositorio.

Ahora, debemos especificar los usuarios que van a tener permiso de modificación al repositorio. Para hacer esto, la forma más sencilla es crear un grupo de usuarios y asignar todos los usuarios que queramos que tengan acceso de modificación al repositorio bajo ese grupo. Por ejemplo, creamos el grupo *usuariossvn* tecleando:

```
$groupadd usuariossvn
```

Y a continuación cambiamos el directorio del repositorio de grupo:

```
$chgrp -R usuariossvn repositorio
```

Con este comando, el directorio repositorio pasa a pertenecer al grupo de usuarios *usuariossvn*. A continuación, podemos editar el fichero /etc/groups para añadir los usuarios que queremos que tengan acceso en escritura al repositorio al grupo *usuariossvn*.

ANOTACIONES

Una vez creado y configurado el repositorio y sus usuarios, podemos obtener un directorio de trabajo local tecleando:

```
svn checkout svn://usuario@dirección/home/jordi/subversion/repositorio
```

Donde el *usuario* representa el usuario que utilizamos para autentificarnos en el repositorio y *dirección* la dirección de la máquina donde nos conectamos, por ejemplo, una dirección IP. Para podernos conectar es importante iniciar el servidor Subversion tecleando:

```
svnverve -d
```

5.7.5. Comandos básicos con Subversion

A continuación veremos algunos de los comandos básicos de Subversion que necesitamos en un proyecto. Prácticamente todos los comandos de Subversion tienen el mismo nombre y sintaxis que los comandos correspondientes en CVS. Esto es así para minimizar la curva de aprendizaje de los usuarios que migran del sistema CVS a Subversión. No entraremos en detalle en aquellos comandos que son de comportamiento idéntico.

Para actualizar nuestro directorio de trabajo, al igual que con CVS usaremos el comando UP (abreviación de *update*):

```
$svn up
```

Durante este proceso de sincronización sólo los archivos que han sido modificados son actualizados. Este comando en ningún caso envía las modificaciones que hayamos realizado en nuestro directorio de trabajo, sino que simplemente nos actualiza nuestro directorio de trabajo con los cambios del repositorio.

Para enviar los cambios que hayamos realizado al repositorio usaremos el comando *commit*:

```
$svs commit
```

Este comando envía los cambios de los archivos modificados y los registra con el comentario indicado. Los comentarios resultan fundamentales

ANOTACIONES

durante el proyecto para entender por qué nosotros u otras personas han introducido los cambios en el proyecto.

Para ver las diferencias de nuestro directorio de trabajo con el repositorio, usaremos el comando *diff*. Tecleamos:

```
$svn diff
```

Este comando mostrará la lista detallada de cambios que hemos introducido en nuestro directorio de trabajo.

Tabla de comandos de Subversion

Tabla 2. Descripción de los principales comandos de Subversion (comando svn)

Comando	Descripción
add	Añade un nuevo archivo o un conjunto de archivos de un directorio al repositorio.
blame (praise, annotate, ann)	Muestra la última revisión del archivo indicando el número de revisión de cada línea del archivo y qué usuario fue el último que la modificó.
cat	Examina las diferencias entre diferentes versiones de un archivo.
checkout (co)	Esta operación es la que realizamos cuando queremos obtener una copia de los archivos del repositorio en nuestra máquina.
cleanup	Limpia nuestro directorio de trabajo de posibles operaciones no finalizadas o archivos bloqueados.
commit	Acción con la que enviamos los cambios que hemos realizado en nuestro directorio de trabajo al repositorio central del servidor.
copy (cp)	Copia un archivo en nuestro directorio de trabajo o repositorio manteniendo el histórico del mismo.
delete (del, remove, rm)	Elimina archivos y directorios permanentemente del repositorio.
diff (di)	Muestra las diferencias de un archivo o grupo de archivos entre nuestro directorio de trabajo al repositorio central del servidor.
export	Exporta una copia del repositorio.
help (?, h)	Muestra la ayuda del programa.
import	Importa archivos al repositorio.
info	Muestra información sobre nuestro directorio de trabajo del repositorio.
list (ls)	Muestra los archivos que hay en el repositorio.
log	Muestra el archivo de registro para un archivo o conjunto de archivos.
merge	Aplica las diferencias entre dos versiones de un archivo al nuestro directorio de trabajo.

ANOTACIONES

Comando	Descripción
mkdir	Crea un directorio en el repositorio.
move (mv, rename, ren)	Mueve o cambia de nombre un archivo en el repositorio.
propdel (pdel, pd)	Elimina una propiedad con su valor de un archivo, conjunto de archivos o directorio.
propedit (pedit, pe)	Edita una propiedad usando un editor externo.
propget (pget, pg)	Muestra el valor de una propiedad en un archivo, conjunto de archivos o directorio.
proplist (plist, pl)	Lista todas las propiedades en un archivo, conjunto de archivos o directorio.
propset (pset, ps)	Establece el valor de una propiedad en un archivo, conjunto de archivos o directorio.
resolved	Elimina el estado de conflicto en archivos y directorios.
revert	Deshace los cambios introducidos en un directorio de trabajo de un archivo.
status (stat, st)	Muestra el estado de archivos y directorios.

5.8. Conclusión

La tabla que viene a continuación contiene una descripción de los principales comandos de Subversion (comando svn).

Los sistemas de control de versiones son la columna vertebral, que propician que grupos de personas trabajen de forma conjunta en el desarrollo de proyectos. Estos sistemas permiten a miles de usuarios en localizaciones geográficas diferentes trabajar colaborativamente en el desarrollo de software libre.

Hemos aprendido a usar tanto la parte cliente, como la servidor, de CVS (*concurrent versions system*) que es el sistema de control de versiones utilizado actualmente en el mundo del software libre. Hemos visto también Subversion, un producto de control de versiones relativamente joven, nació en el año 2001, que tiene como objetivo solventar las principales carencias de CVS y que ya han comenzado a adoptar muchos proyectos en sustitución de CVS.

Con los conocimientos adquiridos de ambos productos, somos perfectamente capaces de poder obtener una copia de trabajo de cualquier proyecto libre, enviar mejoras, o instalar nuestro propio servicio de control de versiones.

ANOTACIONES

5.9. Otras fuentes de referencia e información

Control de versiones con Subversion (versión libre).
http://svnbook.red-bean.com/

Notas de uso de CVS. http://www.cs.columbia.edu/~hgs/cvs/

Open Source Development with CVS (3.ª ed.) (versión libre).
http://cvsbook.red-bean.com/

Source Control.
http://software.ericsink.com/scm/source_control.html

Tarjeta de referencia con los comandos y opciones de CVS.
http://refcards.com/refcards/cvs/index.html

ANOTACIONES

6. Gestión de software

6.1. Introducción

El avance del sistema operativo GNU/Linux, en estos momentos el mayor exponente del software libre, ha hecho evolucionar la manera en la que el software es distribuido en este tipo de entornos. El principal motivo ha sido la necesidad de empaquetar de forma homogénea y consistente el software junto con la base del sistema operativo.

La gran variedad de distribuciones de GNU/Linux dirigidas a distintos fines y usuarios, también está haciendo que se desarrollen nuevos sistemas de distribución de software, que aunque con una base común, difieren en la forma de organizar y relacionar el software entre sí y con sus componentes internos.

6.2. Objetivos

Con la lectura de este capítulo, el lector debe ser capaz de alcanzar los objetivos siguientes:

- Analizar los sistemas de distribución y compresión tradicionales, de forma individual y en conjunción, para posteriormente estudiar los dos principales sistemas de distribución, rpm y deb, entrando no sólo en su manejo, sino en su creación.

- Conocer los principales *front-ends* o software gráfico de gestión de paquetes para acercarnos más a la realidad existente.

El lector, por lo tanto, deberá finalizar el capítulo conociendo y manejando todos los sistemas expuestos, tanto desde el punto de vista funcional como de creación.

ANOTACIONES

6.3. El empaquetador universal: tar

Uno de los principales problemas a la hora de distribuir, copiar o almacenar archivos es el de mantener intactos todos los "metadatos" asociados, como pueden ser la fecha de creación, los permisos de ejecución o el usuario y grupo al que pertenecen. En la distribución también es recomendable que cada elemento de software sea presentado como un archivo único fácilmente identificable.

Para todas estas necesidades existe la utilidad `tar`.

Aunque originalmente pensada para realizar copias de respaldo a cinta ("Tape ARchiver" o archivador en cinta), su uso en entornos Linux ha venido motivado principalmente por su facilidad de manejo y versatilidad en la tarea concreta de realizar copias idénticas, tanto de archivos como estructuras de directorio completas.

La sintaxis básica de la utilidad `tar` es la siguiente:

```
tar modo [opciones] [-f fichero_tar] [ficheros_origen]
```

La utilidad `tar` puede operar tanto sobre ficheros especificados explícitamente, como sobre la entrada y salida estándar del programa (por ejemplo con redirecciones).

Las opciones se componen de un comando principal ("modo de operación") que define si está creando (c), extrayendo (x), listando (t), etc. el contenido del fichero empaquetado.

Con la opción `f nombre_fichero` especificamos que se va a operar sobre un fichero `tar` (y no sobre la entrada/salida estándar).

Añadiendo `v` ("verbose" = detallado) obtenemos un informe de las operaciones que va realizando `tar`, lo que puede ser especialmente útil para detectar errores, o simplemente mantener la calma cuando se manejan grandes cantidades de datos y `tar` parece "no hacer nada".

Veamos un ejemplo de su uso.

Contenido complementario

Utilidad tar
c: crear
x: eXtraer
t: lisTar

Nota

Por motivos de seguridad, los ejemplos no se hacen como superusuario (root), salvo que se diga lo contrario.

ANOTACIONES

Empezaremos creando un directorio temporal:

```
$ mkdir /tmp/test
$ cd /tmp/test
```

Para llenarlo rápidamente de ficheros, podemos hacer:

```
$ cp /dev/null /tmp/test/fichero1
$ cp -r /tmp/test/ /tmp/test/directorio
$ chmod 400 /tmp/test/fichero1
$ ls -l /tmp/test
total 0
drwxr-xr-x  2 user user 60 Nov 17 18:03 directorio/
-r--------  1 user user  0 Nov 17 18:03 fichero1
```

Ahora, supongamos que queremos hacer una copia de seguridad del directorio `test`, podríamos hacer lo siguiente:

```
$ cd /tmp
$ tar cvf /tmp/test.tar test
test/
test/directorio/
test/directorio/fichero1
test/fichero1
$ ls -l /tmp/test.tar
-rw-r--r--  1 user user 10240 nov 17 18:03 /tmp/test.tar
```

Si ahora borrásemos el directorio `test`:

```
$ rm -rf /tmp/test
$ ls test
ls: test: No such file or directory
```

Podríamos restaurarlo completamente desde el fichero `test.tar`:

```
$ cd /tmp
$ tar xvf /tmp/test.tar
test/
test/directorio/
test/directorio/fichero1
test/fichero1
$ ls -l /tmp/test
total 0
drwxr-xr-x  2 user user 60 Nov 17 18:03 directorio/
-r--------  1 user user  0 Nov 17 18:03 fichero1
```

ANOTACIONES

Se puede apreciar que el fichero /tmp/test/fichero1 se ha restaurado con los mismos permisos, usuario y fecha que tenía cuando fue almacenado con tar.

También podemos consultar en cualquier momento cuál es el contenido de un fichero tar:

```
$ tar tvf test.tar
drwxr-xr-x user/user      0 2004-11-17 18:03:49 test/
drwxr-xr-x user/user      0 2004-11-17 18:03:49 test/directorio/
-rw-r--r-- user/user      0 2004-11-17 18:03:49 test/directorio/fichero1
-r-------- user/user   0 2004-11-17 18:03:48 test/fichero1
```

6.3.1. Comprimiendo: gzip

La mayoría de los ficheros almacenados en un ordenador suelen estarlo de tal forma que se puedan leer de disco y utilizar lo más rápidamente posible. Esto implica que aparezca información almacenada en bloques de tamaño fijo (independientemente de si están llenos o no), información de un conjunto de caracteres reducido (por ejemplo, texto legible directamente) y otras formas de almacenamiento en las que no se tiene demasiado en cuenta el espacio que ocupan.

Sin embargo, toda información puede ser codificada de forma que se reduzca el espacio que ocupa sin perder parte de la misma, se trata de ficheros de texto, ejecutables o de cualquier otro tipo, puesto que siempre aparecen en ellos secuencias de bytes que se repiten y son susceptibles de ser comprimidas ("simplificadas").

Una herramienta muy útil para la compresión y descompresión de distintos formatos basados en la codificación Lempel-Ziv es la utilidad gzip/gunzip.

Basada en un documento de 1977 por J. Ziv y A. Lempel sobre un algoritmo universal de compresión secuencial, esta codificación permite comprimir y descomprimir los ficheros directamente mientras se leen byte tras byte.

Al igual que la mayoría de las utilidades del sistema, permite comprimir y descomprimir tanto desde entrada estándar como desde fichero, grabando el resultado en otro fichero o en la salida estándar.

ANOTACIONES

La sintaxis básica es muy simple (en orden: comprimir, descomprimir):

```
gzip nombre_fichero
gunzip nombre_fichero.gz
```

Si no se especifican más opciones, `gzip` comprime el fichero `nombre_fichero` y lo reemplaza con el resultado guardado añadiendo la extensión `.gz` como `nombre_fichero.gz`.

Inversamente, `gunzip` descomprime el fichero `nombre_fichero.gz` y lo reemplaza con el resultado guardado quitando la extensión `.gz` como `nombre_fichero`.

Debido a este funcionamiento y a la capacidad de compresión secuencial, es el complemento ideal para `tar`.

En los ejemplos anteriores sólo se habían manejado ficheros "vacíos", y aun así se puede observar que el fichero .tar ocupa más de 10.000 bytes. Esto es debido a que internamente la información se almacena en bloques de como mínimo 512 bytes (incluido un bloque por fichero con los metadatos) y se graba en bloques de 10.240 bytes.

Normalmente, esto no supone un problema con ficheros de tamaño considerable, sin embargo, es un factor a tener en cuenta si se quiere almacenar gran número de ficheros pequeños.

En todo caso, veamos cómo podemos usar `gzip/gunzip` tanto por separado como junto con `tar`.

Supongamos que ahora, una vez creado `/tmp/test.tar`, queremos hacer que ocupe menos. Sólo hay que ejecutar:

```
$ ls -l /tmp/test.tar
-rw-r--r--  1 user user 10240 Nov 17 18:03 /tmp/test.tar
$ gzip /tmp/test.tar
```

Y obtendremos un fichero comprimido:

```
$ ls -l /tmp/test.tar.gz
-rw-r--r--  1 user user 191 Nov 17 18:03 /tmp/test.tar.gz
```

ANOTACIONES

Obsérvese que al comprimir el fichero NO se ha cambiado ni el usuario, ni los permisos, ni la fecha del mismo. En cambio SÍ que se ha borrado la copia original:

```
$ ls -l /tmp/test.tar
ls: /tmp/test.tar: No such file or directory
```

Veamos si podemos descomprimirlo:

```
$ gunzip /tmp/test.tar.gz
$ ls -l /tmp/test.tar
-rw-r--r--  1 user user 10240 Nov 17 18:03 /tmp/test.tar
$ ls -l /tmp/test.tar.gz
ls: /tmp/test.tar.gz: No such file or directory
```

El fichero .tar.gz ha sido borrado tras la creación del .tar descomprimido.

Ahora, supongamos que queremos crear el fichero .tar.gz, pero sin el paso intermedio de crear el .tar y ocupar tanto espacio en disco (igual a: los datos originales + metadatos + el fichero .tar.gz antes de ser borrado el .tar).

Para estos casos tenemos distintas opciones.

Por un lado se pueden usar redirecciones aprovechando que tanto `tar` como `gzip` pueden trabajar con la salida y entrada estándar:

```
$ cd /tmp
$ tar cv test | gzip > /tmp/test.tar.gz
test/
test/fichero1
test/directorio/
test/directorio/fichero1
$ ls -l /tmp/test.tar.gz
-rw-r--r--  1 user user 184 Nov 17 18:03 /tmp/test.tar.gz
```

O podemos aprovechar directamente una de las opciones de `tar`, la `z` (zip):

```
$ cd /tmp
$ tar czvf /tmp/test.tar.gz test
```

ANOTACIONES

```
test/
test/fichero1
test/directorio/
test/directorio/fichero1
$ ls -l /tmp/test.tar.gz
-rw-r--r--  1 user user 184 Nov 17 18:03 /tmp/test.tar.gz
```

6.3.2. Usando tar, gzip, y unos disquetes

Una situación que puede presentarse al intentar grabar una serie de archivos en un medio como un disquete es que haya algún archivo más grande que la capacidad del disco en sí. Tanto en este caso como para optimizar el espacio en los disquetes, lo más cómodo es poder partir el conjunto de los datos que hay que grabar en trozos del tamaño de cada disquete.

Según esto, se ha establecido la opción `M` (Multi-volumen) de `tar`.

Supongamos que tenemos tres archivos aleatorios de 1 MB cada uno:

```
$ dd bs=1024 count=1024 < /dev/urandom > /tmp/test/fichero1
$ dd bs=1024 count=1024 < /dev/urandom > /tmp/test/fichero2
$ dd bs=1024 count=1024 < /dev/urandom > /tmp/test/fichero3
-rw-r--r--  1 root root 1048576 nov 17 18:03 /tmp/fichero1
-rw-r--r--  1 root root 1048576 nov 17 18:03 /tmp/fichero2
-rw-r--r--  1 root root 1048576 nov 17 18:03 /tmp/fichero3
```

Si intentásemos grabarlos en disquetes de 1.4 MB (1440 KB = 1474560 bytes), habría que elegir entre guardar cada fichero por separado, o comprimir los ficheros por separado y esperar que dos o más cualesquiera ocupasen menos de 1.4 MB:

```
$ gzip /tmp/test/fichero1
$ gzip /tmp/test/fichero2
$ gzip /tmp/test/fichero3
$ ls -l /tmp/test/fichero*.gz
-rw-r--r--  1 root root 1048763 nov 17 18:03 /tmp/fichero1.gz
-rw-r--r--  1 root root 1048763 nov 17 18:03 /tmp/fichero2.gz
-rw-r--r--  1 root root 1048763 nov 17 18:03 /tmp/fichero3.g
```

ANOTACIONES

En este caso los ficheros no se comprimen debido a que se crearon conteniendo sólo secuencias de bytes aleatorios, difícilmente comprimibles.

Para solucionar este problema, podemos recurrir a la opción `M` (multi-volumen) de `tar` comprimiendo directamente en el disquete:

```
$ tar cMf /tmp/floppy/test.tar /tmp/test/fichero*
Prepare el volumen #2 para `test.tar' y pulse intro:
Prepare el volumen #3 para `test.tar' y pulse intro:
```

Con esta opción `tar` intentará grabar el máximo posible de datos en cada disco, posteriormente pedirá que se cambie por uno vacío, y volverá a repetir el proceso hasta que todos los datos estén grabados.

Si en vez de grabar directamente a disco lo que queremos es preparar primero los ficheros, podemos recurrir a la opción `L` (longitud) que nos permitirá especificar la "longitud del medio" (o disco) en múltiplos de 1024 bytes.

Al mismo tiempo, podemos poner más de un fichero `tar` de salida con repetidas opciones `f fichero` (`ff fichero fichero`, `fff fichero fichero fichero`, ...).

```
$ tar cMfff /tmp/test1.tar /tmp/test2.tar /tmp/test3.tar \
>  /tmp/test/fichero* -L 1440
$ ls -l /tmp/test?.tar
-rw-r--r--   1 root root 1474560 nov 17 18:03 test1.tar
-rw-r--r--   1 root root 1474560 nov 17 18:03 test2.tar
-rw-r--r--   1 root root  204800 nov 17 18:03 test3.tar
```

La única limitación de `tar` es que, debido a su uso de `gzip` para comprimir el resultado del propio tar, no permite fraccionar y comprimir los archivos al mismo tiempo (esto es válido para GNU tar 1.14).

6.4. RPM

Con nombre autorreferente "RPM Package Manager" (Gestor de Paquetes RPM), es un formato para la distribución y gestión de paquetes de software entendidos en el sentido de "módulos funcionales indivi-

ANOTACIONES

sibles". La distribución que lo desarrolló e implementó en primer lugar fue Red Hat Linux, a la que siguieron Suse y Mandrake entre otras.

La ventaja de utilizar RPM en vez de ficheros .tar (o .tar.gz) consiste en las opciones adicionales que RPM permite especificar, a la persona que crea el paquete, para ser ejecutadas automáticamente en la instalación, desinstalación, etc. del mismo. De esta forma se simplifica al máximo la tarea al usuario final y, si el paquete está creado correctamente, mantiene una lista de los paquetes instalados y las relaciones entre ellos para evitar que aparezcan conflictos entre los ficheros instalados, o pendientes de instalar.

6.4.1. Trabajar con RPM

Las fases del sistema RPM se pueden dividir en tres partes:

1) El creador del paquete .rpm
2) Una base de datos de paquetes instalados en el sistema del usuario
3) El programa `rpm` en sí

En este caso nos vamos a ocupar del último punto (el programa `rpm`) y de su relación con la base de datos (segundo punto) al instalar y gestionar los distintos paquetes .rpm.

Los sistemas de paquetes han hecho posible la creación de distribuciones GNU/Linux, ya que es una forma sencilla de distribuirlos y guardar una base de datos que les dé consistencia.

Se pueden encontrar en los discos de instalación, en Internet dentro de servidores ftp con copias de las distribuciones, ser facilitados por los autores de los programas, etc.

Suelen estar nombrados en el formato:

```
paquete - versión del software - versión-rpm . arquitectura .rpm
```

Por ejemplo, éstos son paquetes `.rpm`:

```
gcc-3.4.3-3.i386.rpm
apache-1.3.31-7mdk.i586.rpm
bash-3.0-22.x86_64.rpm
perl-5.00503-12.alpha.rpm
```

Lectura complementària

http://rpmfind.net
http://freshmeat.net
ftp://ftp.rediris.es

ANOTACIONES

Los paquetes .rpm son versiones ejecutables (compiladas) de los programas. Dado que los programas bajo licencias libres también distribuyen el código fuente, también existen las versiones en código fuente de los mismos paquetes:

```
gcc-3.4.3-3.src.rpm
apache-1.3.31-7mdk.src.rpm
bash-3.0-22.src.rpm
perl-5.00503-12.src.rpm
```

Como se puede observar, en este caso sólo se ha sustituido el nombre de la arquitectura para la que está destinado el paquete por el texto `src`. Esto es debido a que un paquete de código fuente suele estar preparado para compilarse en cualquier arquitectura. Así, aunque el paquete compilado para `x86_64` tendría problemas para ejecutarse en un entorno PowerPc (`ppc`), el paquete de código fuente puede ser compilado para dar lugar a ambos `x86_64` y `ppc`, que posteriormente se podrán distribuir e instalar directamente en estas arquitecturas.

También pueden llamarse con la cadena `noarch`, que significa estrictamente “independiente de arquitectura”, o `nosrc` que designaría un paquete con sólo las pautas para construir el ejecutable a partir del código fuente pero sin incluirlo.

Entre otras cosas, esto permite que se desarrolle software en una arquitectura y se use en otra con sólo compilar el paquete `src`. O se distribuya sólo en formato src y cada usuario pueda compilarlo a la hora de instalar, sea para optimizarlo para su sistema o con el fin de asegurarse de que la versión ejecutable corresponda realmente al código fuente obtenido del programa.

6.4.2. Instalación

La instalación de un paquete en versión ejecutable no supone mayor dificultad con la opción `-i` de `rpm`:

```
$ rpm -ivh nc-1.10-18mdk.i586.rpm
Preparing... #################### [100%]
   1:nc      #################### [100%]
```

Contenido complementario

Utilidad rpm
i: instalar
u: actualizar (Update)
q: consultar (Query)
e: eliminar
V: verificar

ANOTACIONES

La opción `v` (minúscula) hace que se muestren los pasos que se van dando. La `h` muestra las barras de progreso (#).

Como se ha dicho antes, RPM permite asegurarse de que no hay conflictos entre paquetes. Una de las formas de conseguirlo es especificando en cada paquete qué otros harán falta para su correcto funcionamiento. Esto es lo que se llaman "dependencias" (los paquetes "dependen" unos de otros).

Junto con la base de datos listando los paquetes instalados, esto hace que al instalar un paquete sea posible detectar la falta de alguna dependencia (otro paquete) necesaria para que funcione correctamente:

```
$ rpm -ivh gaim-1.0.3-1mdk.i586.rpm
error: Failed dependencies:
       libgaim-remote = 1:1.0.3-1mdk is needed by gaim-1.0.3-1mdk
```

Las dependencias pueden ser de una versión concreta o de una comparación booleana (por ejemplo: >= , que significaría "mayor o igual a la pedida").

Para resolverlas será necesario descargar el paquete que falta e instalarlo antes o junto al paquete que lo necesita:

```
$ rpm -ivh gaim-1.0.3-1mdk.i586.rpm \
>  libgaim-remote0-1.0.3-1mdk.i586.rpm
Preparing...                ##################### [100%]
   1:libgaim-remote0         ##################### [ 50%]
   2:gaim                    ##################### [100%]
```

En este caso `rpm` se encarga del orden correcto en el que deben ser instalados cada uno de los paquetes.

Si sólo se desea comprobar si es posible o no instalar un paquete determinado, pero sin instalarlo realmente, se puede usar la opción --`test`:

```
$ rpm -ivh --test gaim-1.0.3-1mdk.i586.rpm \
>  libgaim-remote0-1.0.3-1mdk.i586.rpm
Preparing...                ##################### [100%]
$ echo $?
  0
```

ANOTACIONES

El valor de retorno de esta opción se puede usar para automatizar las decisiones de instalación:

```
$ rpm -ivh --test gaim-1.0.3-1mdk.i586.rpm
error: Failed dependencies:
  libgaim-remote = 1:1.0.3-1mdk is needed by gaim-1.0.3-1mdk
  libgaim-remote.so.0 is needed by gaim-1.0.3-1mdk
$ echo $?
  1
```

Si fallan las dependencias, también se puede elegir ignorar este fallo e instalar el paquete de todas formas con la opción `--nodeps`, aunque probablemente luego no funcione:

```
$ rpm -ivh --nodeps gaim-1.0.3-1mdk.i586.rpm
Preparing...                ##################### [100%]
  2:gaim                  ##################### [100%]
```

En caso de que dos paquetes entrasen en conflicto:

```
$ rpm -ivh --nodeps gaim-1.0.2-1mdk.i586.rpm
Preparing...                ##################### [100%]
package gaim-1.0.3-1mdk (which is newer than gaim-1.0.2-1mdk) is already installed
file /usr/bin/gaim from install of gaim-1.0.2-1mdk conflicts with file from package gaim-1.0.3-1mdk
file /usr/bin/gaim-remote from install of gaim-1.0.2-1mdk conflicts with file from package gaim-1.0.3-1mdk
file /usr/lib/gaim/autorecon.so from install of gaim-1.0.2-1mdk conflicts with file from package gaim-1.0.3-1mdk
file /usr/lib/gaim/docklet.so from install of gaim-1.0.2-1mdk conflicts with file from package gaim-1.0.3-1mdk
...
```

Nota

Advertimos que esta forma de instalación ignora las recomendaciones de rpm y, por lo tanto, del autor del programa o del encargado de crear y testear el paquete. Esto puede conducir a un sistema inestable y distintos conflictos entre programas, que normalmente no serán tomados en cuenta por los desarrolladores.

Se puede buscar alguno compatible con el instalado (o actualizar éste), o forzar la instalación del nuevo paquete independientemente de los avisos. En este caso, es probable que el paquete no funcione correctamente, que sobreescriba partes de otro paquete, etc. Hay distintas opciones que permiten saltarse distintas partes (`--replacepkgs`, `--replacefiles`, `--oldpackage`) de la comprobación, o saltarse todas con `--force`:

```
$ rpm -ivh --nodeps --force gaim-1.0.3-1mdk.i586.rpm
Preparing...                ##################### [100%]
    2:gaim                  ##################### [100%]
```

ANOTACIONES

6.4.3. Actualización

Para actualizar un paquete, lo más obvio sería desinstalar la versión antigua e instalar en su lugar la nueva. Sin embargo, esto puede romper algunas dependencias de paquetes que son imprescindibles para el que intentamos actualizar.

Esto se puede evitar con la opción `-U` ("Update" = actualizar) de `rpm`.

Su uso es igual que el de la `-i` para instalar, con la diferencia de que si alguna de las dependencias que entran en conflicto con la instalación es del mismo paquete pero con versión anterior, éste será eliminado y reemplazado por el nuevo de forma automática. Primero instalamos una versión anterior:

```
$ rpm -ivh gaim-1.0.2-1mdk.i586.rpm \
> libgaim-remote0-1.0.2-1mdk.i586.rpm
Preparing...                ##################### [100%]
   1:libgaim-remote0         ##################### [ 50%]
   2:gaim                    ##################### [100%]
$ rpm -q gaim libgaim-remote0
gaim-1.0.2-1mdk
libgaim-remote0-1.0.2-1mdk
```

Y posteriormente actualizamos a una versión superior:

```
$ rpm -Uvh gaim-1.0.3-1mdk.i586.rpm \
> libgaim-remote0-1.0.3-1mdk.i586.rpm
Preparing...                ##################### [100%]
   1:libgaim-remote0         ##################### [ 50%]
   2:gaim                    ##################### [100%]
$ rpm -q gaim libgaim-remote0
gaim-1.0.3-1mdk
libgaim-remote0-1.0.3-1mdk
```

Se puede apreciar que rpm detecta que la versión nueva es una actualización de la anterior y no presenta mensaje de error por el conflicto entre paquetes y ficheros instalados.

Durante una actualización se mantienen los ficheros de configuración de los paquetes salvo que los nuevos sean muy diferentes y con los

antiguos posiblemente no funcionasen los paquetes, en cuyo caso rpm hace una copia de seguridad (añadiendo la extension .rpmsave al nombre del fichero) e instala en su lugar la nueva versión del mismo. En otras ocasiones, cuando el fichero antiguo es compatible con la nueva versión del programa pero hay un fichero nuevo con otras opciones, es el nuevo el que se salva aparte (añadiendo la extensión .rpmnew)

6.4.4. Consulta

La base de datos de paquetes instalados, así como toda la información relativa al paquete, pueden ser consultados con la opción `-q` ("Query" = consultar) de `rpm`:

```
$ rpm -q gaim
gaim-1.0.3-1mdk
```

Para ver una lista de todos los paquetes instalados, podemos usar la opcion `a` ("All" = todos):

```
$ rpm -qa
libltdl3-1.5.6-4mdk
termcap-11.0.1-8mdk
libofx-0.6.6-2mdk
...
```

Dado que esta lista suele ser bastante larga, para ver los paquetes instalados es más cómodo usar:

```
$ rpm -qa | less
```

Para consultar los datos de un paquete `-i` (información):

```
$ rpm -qi gaim
Name        : gaim                        Relocations: (not relocatable)
Version     : 1.0.3                            Vendor: Mandrakesoft
Release     : 1mdk                         Build Date: vie 12 nov 2004 23:29:22 CE
Install Date: sáb 20 nov 2004 08:28:52 CET     Build Host: n4.mandrakesoft.com
Group       : Networking/Instant messaging   Source RPM: gaim-1.0.3-1mdk.src.rpm
Size        : 8514053                         License: GPL
Signature   : DSA/SHA1, vie 12 nov 2004 23:40:27 CET, Key ID dd684d7a26752624
```

ANOTACIONES

```
Packager    : Pascal Terjan <pterjan@mandrake.org>
URL         : http://gaim.sourceforge.net/
Summary     : A GTK+ based multiprotocol instant messaging client
Description :
Gaim allows you to talk to anyone using a variety of messaging
protocols, including AIM (Oscar and TOC), ICQ, IRC, Yahoo!,
MSN Messenger, Jabber, Gadu-Gadu, Napster, and Zephyr.  These
protocols are implemented using a modular, easy to use design.
To use a protocol, just load the plugin for it.

Gaim supports many common features of other clients, as well as many
unique features, such as perl scripting and C plugins.
Gaim is NOT affiliated with or endorsed by AOL.
```

Donde pueden verse diferentes datos relativos al paquete.

Si es de un paquete no instalado en el sistema, y que por lo tanto no está en la base de datos, teniendo el fichero .rpm podemos hacer referencia a él con `-p` (fichero):

```
$ rpm -qp gaim-1.0.3-1mdk.i586.rpm
gaim-1.0.3-1mdk
```

La lista de ficheros contenidos en el paquete puede consultarse con `-l`. Añadiendo `-v` se muestran también los datos de cada fichero:

```
$ rpm -qlv gaim
-rwxr-xr-x    1 root    root          853644 nov 12 23:29 /usr/bin/gaim
-rwxr-xr-x    1 root    root            7608 nov 12 23:29 /usr/bin/gaim-remote
drwxr-xr-x    2 root    root               0 nov 12 23:29 /usr/lib/gaim
-rwxr-xr-x    1 root    root            8676 nov 12 23:29 /usr/lib/gaim/autorecon.so
...
```

También se puede solicitar sólo determinado “tipo” de ficheros, como documentación `-d`:

```
$ rpm -qld coreutils
/usr/share/doc/coreutils-5.2.1/README
```

O sólo los de configuración:

```
$ rpm -qlc coreutils
/etc/DIR_COLORS
/etc/pam.d/su
```

ANOTACIONES

De esta forma se pueden automatizar tareas relacionadas con la documentación (p. ej. creación de índices) o la configuración (p. ej. copia de seguridad) de los paquetes.

6.4.5. Desinstalación

La desinstalación de un paquete se lleva a cabo con la opción -e.

Igual que en la instalación, si se intenta quitar un paquete del que dependen otros, `rpm` avisará con un mensaje de error:

```
$ rpm -ev libgaim-remote0
error: Failed dependencies:
        libgaim-remote = 1:1.0.3-1mdk is needed by (installed) gaim-1.0.3-1mdk
        libgaim-remote.so.0 is needed by (installed) gaim-1.0.3-1mdk
```

Y, al igual que antes, es suficiente con especificar a la vez todos los paquetes que se van a desinstalar:

```
$ rpm -ev libgaim-remote0 gaim
```

6.4.6. Verificación

Contenido complementario

Tipos de verificaciones
S: tamaño (*size*) del fichero
M: **m**odo (permisos)
5: suma MD**5**
D: **d**ispositivo (mayor/menor)
L: destino del **enl**ace
U: **u**suario (propietario)
G: **g**rupo (propietario)
T: fecha (*time*) del fichero

Otra de las funciones interesantes de RPM es la posibilidad de verificar el estado de cada fichero instalado para detectar posibles modificaciones.

En cada paquete se guarda la información sobre el tamaño, suma de comprobación, propietarios, etc. de cada fichero. A la hora de instalarlo, estos datos también son almacenados en la base de datos de RPM, de forma que luego pueden ser utilizados con el fin de compararlos con los mismos datos obtenidos de los ficheros presentes en el sistema y verificar que éstos no han sido modificados.

La sintaxis básica se centra en la opción `--verify` (`-V`) de rpm:

```
rpm -V coreutils
S.5....T c /etc/pam.d/su
```

ANOTACIONES

Por cada fichero distinto a su correspondiente del paquete original, se muestra una línea con la lista de los cambios detectados y el tipo de fichero: configuración, documentación, *ghost* (no incluido pero creado), licencia, *readme* ('leeme'). No se muestran, sin embargo, diferencias para ficheros no instalados originalmente con el paquete.

Otra forma de verificación consiste en la validación de la firma OpenPGP (gpg del paquete en sí por medio de la opción –checksig:

```
# rpm --checksig X11R6-contrib-6.7.0-4.2.101mdk.i586.rpm
X11R6-contrib-6.7.0-4.2.101mdk.i586.rpm:
    Header SHA1 digest: OK (caa530fe83f24cbdeefbfbfb0db53f21cd860e07)
    MD5 digest: OK (023c21cf6ef0f6f0841ffa0b9010ac43)
    V3 DSA signature: OK, key ID 22458a98
```

Para esto la parte pública de la clave gpg con la que esté firmado el paquete debe estar incluida en el anillo de claves del usuario que ejecute rpm, o en el especificado por la macro %_gpg_path.

6.4.7. Creación de paquetes y gestión de parches

La creación de paquetes se centra en la utilidad rpmbuild y un fichero de .spec que especifique los metadatos y pasos a dar en la compilación, instalación y desinstalación de cada paquete.

Quizá la forma más simple sea la creación de un paquete binario a partir de otro con el código fuente:

```
$ rpmbuild --rebuild -v gaim-1.0.1-0.src.rpm
Installing gaim-1.0.1-0.src.rpm
Executing(%prep): [...]
+ tar -xvvf - [...]
Executing(%build): [...]
+ ./configure
checking for [...]
+ make [...]
Executing(%doc): [...]
Wrote: /usr/src/RPM/RPMS/i586/gaim-1.0.1-0.i586.rpm
Wrote: /usr/src/RPM/RPMS/i586/gaim-devel-1.0.1-0.i586.rpm
Executing(%clean): [...]
```

ANOTACIONES

Aquí vemos los pasos que da rpmbuild para crear el paquete final:

1) Sección `%prep` del fichero .spec. En esta sección se suele descomprimir el código fuente y pasar todos los parches que pudiesen ser necesarios.

2) Sección `%build` del fichero .spec. Aquí es donde se compila el paquete. En caso de tener *scripts* de configuración, éstos se ejecutan.

3) Sección `%install` del fichero .spec. Se instala el programa compilado en un directorio temporal.

4) Sección `%doc` del fichero .spec. Se instala la documentación correspondiente.

5) Se empaqueta la instalación temporal en un fichero .rpm.

6) Se crea un paquete adicional con los encabezados (.h) de los ficheros de código fuente para permitir el desarrollo y/o compilación de paquetes basados en éste.

7) Sección `%clean` del fichero .spec. Para terminar, se eliminan todos los ficheros temporales (incluido el código fuente instalado).

No es obligatorio efectuar estos pasos todos seguidos de una vez. Para ello, rpmbuild ofrece las opciones `-bp`, `-bc` y `-bi` que corresponden a parar la ejecución tras las fases de `%prep`, `%build` e `%install` respectivamente.

De esta forma se puede continuar la instalación tras realizar las oportunas modificaciones en cada fase. Por ejemplo, se puede ejecutar sólo `%prep`, pasar algún parche adicional al código fuente, y seguir la instalación desde ahí.

Para la creación de paquetes, rpmbuild ofrece las opciones `-bs`, `-bb` y `-ba`, con las que se pueden crear paquetes de código fuente, binarios, o de ambos tipos respectivamente.

ANOTACIONES

Como paso adicional, al igual que antes hemos podido comprobar la firma de los paquetes, ahora la podemos crear por medio de la opción --addsign:

```
$ rpm --addsign -v X11R6-contrib-6.7.0-4.2.101mdk.i586.rpm
Enter pass phrase:
Pass phrase is good.
X11R6-contrib-6.7.0-4.2.101mdk.i586.rpm:
```

La configuración de la clave que hay que utilizar se puede especificar en /etc/rpm/macros o en ~/.rpmmacros con la sintaxis:

```
%_signature gpg
%_gpg_path /etc/rpm/.gpg
%_gpg_name Nombre Usuario <nombre@dominio.com>
%_gpgbin /usr/bin/gpg
```

Cogiendo primero la clave del anillo del usuario en curso y posteriormente del especificado en %_gpg_path.

Se puede comprobar su validez otra vez con –checksig:

```
$ rpm --checksig -v X11R6-contrib-6.7.0-4.2.101mdk.i586.rpm
X11R6-contrib-6.7.0-4.2.101mdk.i586.rpm:
    Header V3 DSA signature: NOKEY, key ID 0f219629
    Header SHA1 digest: OK (caa530fe83f24cbdeefbfbfb0db53f21cd860e07)
    MD5 digest: OK (023c21cf6ef0f6f0841ffa0b9010ac43)
    V3 DSA signature: NOKEY, key ID 0f219629
```

Donde se observa que la nueva firma ha sustituido a la anterior. Esto es, un paquete sólo puede estar firmado con una única firma al mismo tiempo.

6.4.8. Para finalizar con RPM

La configuración de RPM se define en el fichero /etc/rpm/macros y en el ~/.rpmmacros de cada usuario como modificaciones a las macros por defecto.

Para consultar las macros y demás configuración efectivas en cada momento se puede usar la opción --showrc:

ANOTACIONES

```
$ rpm --showrc
ARCHITECTURE AND OS:
build arch            : i586
compatible build archs: athlon i686 i586 i486 i386 noarch
build os              : Linux
compatible build os's : Linux
install arch          : athlon
install os            : Linux
compatible archs      : athlon i686 i586 i486 i386 noarch
compatible os's       : Linux

RPMRC VALUES:
macrofiles          : /usr/lib/rpm/macros:/usr/lib/rpm/athlon-linux/macros:/etc/rpm/macros.specspo:/etc/
rpm/macros.prelin
k:/etc/rpm/macros.solve:/etc/rpm/macros.up2date:/etc/rpm/macros:/etc/rpm/athlon-linux/macros:/etc/
rpm/macros.jpackage:~/.rp
mmacros
optflags              : -O2 -fomit-frame-pointer -pipe -march=athlon %{debugcflags}

Features supported by rpmlib:
    rpmlib(VersionedDependencies) = 3.0.3-1
        PreReq:, Provides:, and Obsoletes: dependencies support versions.
[...]
========================
-14: GNUconfigure(MCs:)
  CFLAGS="${CFLAGS:-%optflags}" ; export CFLAGS;
  LDFLAGS="${LDFLAGS:-%{-s:-s}}"  ; export LDFLAGS;
  %{-C:_mydir="`pwd`"; %{-M: %{__mkdir} -p %{-C*};} cd %{-C*}}
  dirs="`find ${_mydir} -name configure.in -print`"; export dirs;
    for coin in `echo ${dirs}`
do
  dr=`dirname ${coin}`;
if test -f ${dr}/NO-AUTO-GEN; then
 :
else
[...]
======================== active 321 empty ()
```

ANOTACIONES

Para modificar cualquiera de estas macros y opciones de configuración, es suficiente con añadirlas a uno de los ficheros de configuración vigentes durante la ejecución de `rpm`.

En raras ocasiones (por ejemplo, tras un uso "creativo" de rpm) es posible que la base de datos contenga inconsistencias en relación con los datos de los paquetes instalados. Un síntoma puede ser, por

ejemplo, que al intentar listar los paquetes instalados con rpm -qa, se muestren una serie de éstos pero, al llegar a un punto, rpm no pueda continuar.

Para solucionarlo, suele ser suficiente re-crear los índices de la base de datos a partir del listado de paquetes instalados con la opción --rebuilddb:

```
$ rpm --rebuilddb
```

Si con esto no fuese suficiente, o directamente se quisiese limpiar la base de datos de rpm, se puede recurrir a la opción --initdb:

```
$ rpm --initdb
```

Otra forma de obtener datos de los paquetes instalados es por medio de la opción --queryformat, que permite especificar los datos concretos que nos interesan de cada paquete:

```
$ rpm -q --queryformat "%{NAME} %{OS}\n%{LICENSE}\n" gaim
gaim linux
GPL
```

Una lista de todos los campos que se pueden usar en la cadena de formato se obtiene con la opción -querytags:

```
$ rpm -querytags
ARCH
ARCHIVESIZE
[...]
VERSION
XPM
```

6.5. DEB

El formato de paquetes utilizado por Debian y distribuciones derivadas es el .deb. Se diferencia de los .rpm, aparte de en el formato del paquete en sí, en ciertas opciones de dependencias y en que está pensado directamente para ser utilizado por gestores de paquetes

ANOTACIONES

como `apt`. Esta última diferencia va perdiendo importancia según se mejoran los gestores de paquetes para `rpm`.

6.5.1. APT, DPKG, DSELECT, APTITUDE, CONSOLE APT, etc.

Los paquetes en sí son los .deb. Aunque el formato puede variar entre distribuciones [consultar `deb(5)`], son ficheros que contienen un fichero .tar.gz con el contenido en sí y unos metadatos de control.

La gestión de estos paquetes se realiza por medio del software dpkg, que mantiene una base de datos de metadatos de los paquetes instalados y se encarga de instalarlos y desinstalarlos.

Mientras tanto, la gestión automatizada de dependencias se realiza por medio de la utilidad apt, que permite autodetectar, descargar e instalar automáticamente todos los elementos necesarios para el correcto funcionamiento de un programa.

Tanto dpkg como apt tienen sus correspondientes *front-ends* llamados dselect y aptitude respectivamente. Aunque directamente dpkg y apt ofrecen mayor flexibilidad y número de opciones, el manejo de listas de paquetes se hace más fácil al poder navegar a través de estas listas y operar sobre cada paquete en concreto.

Existen otros programas capaces de realizar estas tareas, incluso, gráficos, como Synaptic, aunque no son oficiales del proyecto Debian.

6.5.2. APT, comandos básicos

Apt se compone de una serie de utilidades encargadas por un lado de la gestión de dependencias y por el otro de los repositorios y listados de paquetes.

Estas utilidades son:

`apt-get`: encargado de descargar/instalar los paquetes.

`apt-cache`: ofrece un acceso a los listados de paquetes.

ANOTACIONES

`apt-cdrom`: permite añadir repositorios de paquetes en CD o DVD.

`apt-ftparchive`: permite añadir repositorios accesibles por ftp.

Aparte de estas utilidades, la configuración de repositorios se almacena en:

```
/etc/apt/sources.list
```

descrita en la entrada de manual `source.list(5).`

Antes de nada hay que obtener una lista de las últimas versiones de los paquetes disponibles en los distintos repositorios que tengamos configurados. Estos repositorios son conjuntos de paquetes almacenados en lugares accesibles por red, cuya estructura es reconocible por el software instalador.

```
# apt-get update
Get:1 http://non-us.debian.org stable/non-US/main Packages [44.5kB]
Get:2 http://http.us.debian.org stable/main Packages [1773kB]
[...]
Get:17 http://security.debian.org stable/updates/non-free Packages [1221B]
Get:18 http://security.debian.org stable/updates/non-free Release [114B]
Fetched 2151kB in 31s (67.8kB/s)
Reading Package Lists... Done
Building Dependency Tree... Done
```

Una vez actualizada la base de datos, podemos proceder a instalar algún paquete que nos interese con `apt-get install`:

```
# apt-get install aptitude
Reading Package Lists... Done
Building Dependency Tree... Done
```

... tras analizar la base de datos de dependencias, es posible que se necesiten paquetes adicionales para que el software que nos interesa funcione correctamente:

```
The following extra packages will be installed:
  libsigc++0
```

ANOTACIONES

```
The following NEW packages will be installed:
  aptitude libsigc++0
0 packages upgraded, 2 newly installed, 0 to remove and 19  not upgraded.
Need to get 939kB of archives. After unpacking 3310kB will be used.
Do you want to continue? [Y/n]
```

... si no tenemos nada en contra de la detección realizada por `apt`, el siguiente paso es descargar los paquetes desde el repositorio más adecuado:

```
Get:1 http://http.us.debian.org stable/main libsigc++0 1.0.4-3 [28.0kB]
Get:2 http://http.us.debian.org stable/main aptitude 0.2.11.1-4 [910kB]
Fetched 939kB in 14s (64.5kB/s)
```

... y posteriormente instalarlos:

```
Selecting previously deselected package libsigc++0.
(Reading database ... 5275 files and directories currently installed.)
Unpacking libsigc++0 (from .../libsigc++0_1.0.4-3_i386.deb) ...
Selecting previously deselected package aptitude.
Unpacking aptitude (from .../aptitude_0.2.11.1-4_i386.deb) ...
Setting up libsigc++0 (1.0.4-3) ...
Setting up aptitude (0.2.11.1-4) ...
```

Además de instalar, también podemos actualizar todos los paquetes a las últimas versiones disponibles en los repositorios con `apt-get upgrade`:

```
# apt-get upgrade
Reading Package Lists... Done
Building Dependency Tree... Done
18 packages upgraded, 0 newly installed, 0 to remove and 0  not upgraded.
Need to get 6778kB of archives. After unpacking 419kB will be used.
Do you want to continue? [Y/n]
```

... si aceptamos la propuesta de actualización de `apt-get`, pasará a descargar los paquetes necesarios:

```
Get:1 http://security.debian.org stable/updates/main gzip 1.3.2-3woody3 [62.1kB]
Get:2 http://http.us.debian.org stable/main bsdutils 1:2.11n-7 [39.5kB]
[...]
```

ANOTACIONES

... y a instalarlos:

```
(Reading database ... 6842 files and directories currently installed.)
Preparing to replace bsdutils 1:2.11n-4 (using .../bsdutils_1%3a2.11n-7_i386.deb) ..
Unpacking replacement bsdutils ...
Setting up bsdutils (2.11n-7) ...
[...]
```

De esta forma podemos mantener siempre actualizados los programas en la última versión disponible en los repositorios elegidos.

Una forma más cómoda de gestionar los paquetes es con el *front-end* oficial de apt llamado aptitude:

Figura 1

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1           Will use 423kB of disk spa DL Size: 72
--- Upgradable Packages
--- Installed Packages
--- Not Installed Packages
--- Virtual Packages
--- Tasks


A newer version of these packages is available.
```

Aquí encontraremos una lista de los paquetes instalados e instalables de la que podremos elegir cualquiera de ellos para instalar nuevos y actualizar o desinstalar los ya presentes.

Figura 2

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1           Will use 423kB of disk spa DL Size: 72
--\ Upgradable Packages
  --- base - The Debian base system
  --- doc - Documentation and specialized programs for viewing doc
  --- editors - Text editors and word processors
  --- libs - Collections of software routines
  --- mail - Programs to write, send, and route email messages
  --- net - Programs to connect to and provide various services

Packages in the 'base' section are part of the initial system
installation.
```

ANOTACIONES

Los paquetes aparecen ordenados por temas para facilitar la localización de aquel que nos interese.

Dentro de la lista de paquetes, al posicionarnos sobre cualquiera de ellos se muestra la información pertinente en la mitad inferior de la pantalla:

Figura 3

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1            Will use 423kB of disk spa DL Size: 72
--\ Upgradable Packages
  --\ base - The Debian base system
    --\ main - The main Debian archive
iu     bsdutils                                1:2.11n-4 1:2.11n-7
iu     debianutils                             1.16      1.16.2wood
iu     gzip                                    1.3.2-3   1.3.2-3woo
iu     libc6                                   2.2.5-6   2.2.5-11.5
Miscellaneous utilities specific to Debian.
Debianutils includes installkernel mkboot mktemp readlink
run-parts savelog
sensible-editor sensible-pager tempfile which.
```

Al seleccionar un paquete, se presenta una descripción y una serie de metadatos relacionados, incluidas las dependencias y sugerencias para su instalación.

Figura 4

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1            Will use 423kB of disk spa DL Size: 72
debianutils - Miscellaneous utilities specific to Debian.
Debianutils includes installkernel mkboot mktemp readlink run-par
sensible-editor sensible-pager tempfile which.
  Essential: yes
  Priority: required
  Section: base
  Maintainer: Guy Maor <maor@debian.org>
  Compressed size: 32.9k
  Uncompressed size: 184k
  Source Package: debianutils
  --\ PreDepends
    --\ libc6 (>= 2.2.4-4)
pi      2.2.5-11.5
id      2.2.5-6
```

Del mismo modo se listan los paquetes virtuales. Esto es, paquetes "de funcionalidad" agrupando distintos paquetes equivalentes entre

sí que ofrezcan una funcionalidad determinada al sistema (servidor DNS, servidor HTTP, etc):

Figura 5

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1            Will use 423kB of disk spa DL Size: 72
--- Virtual Packages
--\ Tasks
  --- Development
  --- End-user
  --- Localization
  --- Miscellaneous
  --- Servers

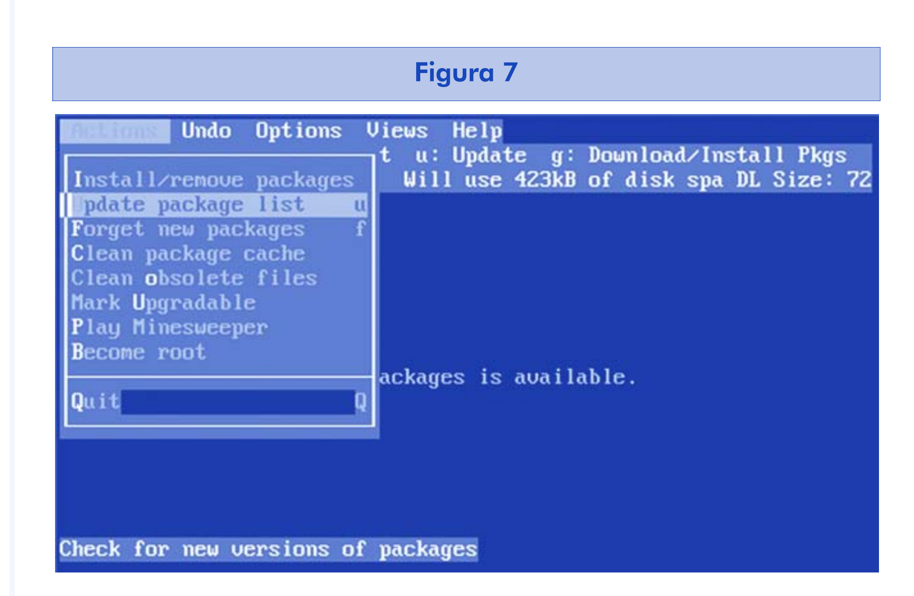
Tasks are groups of packages which provide an easy way to select a
predefined set of packages for a particular purpose.
```

Figura 6

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1            Will use 423kB of disk spa DL Size: 72
  --\ Servers
    --\ DNS server
p     bind9                                  <none>     1:9.2.1-2.
p     bind9-doc                              <none>     1:9.2.1-2.
p     dlint                                  <none>     1.4.0-4
p     dnsutils                               <none>     1:9.2.1-2.
p     lwresd                                 <none>     1:9.2.1-2.
Internet Domain Name Server
The Berkeley Internet Name Domain (BIND) implements an Internet
domain
name server.  BIND is the most widely-used name server software on
the
Internet, and is supported by the Internet Software Consortium,
www.isc.org.
```

Para facilitar las tareas de mantenimiento, aptitude presenta un menú (accesible con F10), desde el que se pueden realizar distintas tareas de administración, incluido algún entretenimiento alternativo como mirar la barra de progreso en instalaciones largas.

Figura 7



También desde este menú se pueden configurar varios aspectos del comportamiento de `apt`:

Figura 8

```
 Actions  Undo  Options  Views  Help
f10: Menu  ?: Help  q: Quit  u: Update  g: Download/Install Pkgs
aptitude 0.2.11.1          Will use 423kB of disk spa DL Size: 72
--- Upgradable Packages
--- Installed Packages

 [X] Automatically resolve dependencies of a package when it is s
 [X] Automatically fix broken packages before installing or remov
 [X] Install Recommended packages automatically
 [X] Install Suggested packages automatically
 [X] Remove unused packages automatically
   Automatically remove unused packages matching this filter
                              [ Ok ]                          [ C
```

6.5.3. DPKG

El programa encargado de la gestión de la base de datos y de la instalación de paquetes es `dpkg`.

Para la funcionalidad de instalación, normalmente se llamará desde apt, dada la comodidad que ofrece la detección automática de dependencias.

Sin embargo, `dpkg` ofrece otras funcionalidades adicionales relacionadas con la base de datos de paquetes instalados.

ANOTACIONES

En la base de datos un paquete puede estar en uno de los siguientes estados:

`not-installed`: sin instalar en el sistema.

`half-installed`: a medio instalar (la instalación posiblemente se ha interrumpido).

`unpacked`: copiado al sistema pero sin configurar.

`half-configured`: a medio configurar (se ha comenzado a configurar pero sin terminar).

`installed`: instalado correctamente.

`config-files`: desinstalado pero sin borrar los ficheros de configuración.

Normalmente todos los paquetes son gestionados por dpkg, excepto los marcados como “hold”.

Para ver un listado de todos los paquetes instalados, junto con sus estados y versiones, podemos utilizar la opción `-l` de `dpkg`:

```
$ dpkg -l
Desired=Unknown/Install/Remove/Purge/Hold
| Estado=No/Instalado/Config-files/Unpacked/Failed-config/Half-installed
|/ Err?=(none)/Hold/Reinst-required/X=both-problems (Status,Err: mayúsc.=malo)
||/ Nombre         Versión        Descripción
+++-==============-==============-============================================
ii  adduser        3.47           Add and remove users and groups
ii  alien          8.05-3         install non-native packages with dpkg
ii  apt            0.5.4          Advanced front-end for dpkg
ii  apt-utils      0.5.4          APT utility programs
ii  aptitude       0.2.11.1-4     curses-based apt frontend
[...]
```

La primera columna indica el estado en el que se desearía tener el paquete, mientras que la segunda indica el estado actual en el sistema. En la tercera columna se indican los errores de instalación, ésta se encuentra vacía al no existir errores.

ANOTACIONES

Aunque es recomendable hacerlo desde apt, también se pueden instalar paquetes por separado desde dpkg con la opción `-i`:

```
$ dpkg -i /var/cache/apt/archives/alien_8.05_all.deb
Seleccionando el paquete alien previamente no seleccionado.
(Leyendo la base de datos ...
7225 ficheros y directorios instalados actualmente.)
Desempaquetando alien (de .../archives/alien_8.05_all.deb) ...
Configurando alien (8.05) ...
```

Así como desinstalarlos con la opción `-r`:

```
$ dpkg -r alien
(Leyendo la base de datos ...
7258 ficheros y directorios instalados actualmente.)
Desinstalando alien ...
```

Un paso intermedio de la instalación consistente en sólo la copia de los ficheros al sistema se puede conseguir con la opción `--unpack`:

```
$ dpkg --unpack /var/cache/apt/archives/alien_8.05_all.deb
Seleccionando el paquete alien previamente no seleccionado.
(Leyendo la base de datos ...
7225 ficheros y directorios instalados actualmente.)
Desempaquetando alien (de .../archives/alien_8.05_all.deb) ...
```

Para configurarlo más tarde con la opción `--configure`:

```
$ dpkg --configure alien
Configurando alien (8.05) ...
```

Los *scripts* de configuración de los paquetes tienen otra aplicación interesante. Aparte de permitir la configuración interactiva en el momento de la instalación de cada paquete, esto añade interesantes posibilidades de configuración al poder "reconfigurar" un paquete ya instalado:

```
$ dpkg-reconfigure locales
```

Por ejemplo, con el anterior comando podemos cambiar la configuración de idioma instalada en el sistema sin tener que preocuparnos

ANOTACIONES

de conocer los ficheros exactos de configuración ni la sintaxis de los mismos.

6.5.4. Creación de paquetes deb y gestión de parches

Una parte de `dpkg` es la gestión de paquetes .deb, que se realiza por medio de `dpkg-deb` (se puede llamar directamente como `dpkg`).

Para listar los ficheros que contiene un paquete, está la opción `-c` (`--contents`):

```
$ dpkg -c alien_8.05-2_all.deb
drwxr-xr-x root/root         0 2004-11-17 18:03:37 ./
drwxr-xr-x root/root         0 2004-11-17 18:03:33 ./usr/
drwxr-xr-x root/root         0 2004-11-17 18:03:33 ./usr/bin/
-rwxr-xr-x root/root     13482 2002-04-01 18:37:26 ./usr/bin/alien
drwxr-xr-x root/root         0 2004-11-17 18:03:34 ./usr/share/
drwxr-xr-x root/root         0 2004-11-17 18:03:33 ./usr/share/alien/
drwxr-xr-x root/root         0 2004-11-17 18:03:33 ./usr/share/alien/patches/
[...]
```

Una información general sobre el paquete se obtiene con `-I` (`--info`):

```
$ dpkg -I alien_8.05-2_all.deb | less
 paquete debian nuevo, versión 2.0.
 tamaño 112716 bytes: archivo de control= 1511 bytes.
     518 bytes,    19 líneas      control
    1989 bytes,    27 líneas      md5sums
     249 bytes,     9 líneas   *  postinst             #!/bin/sh
     192 bytes,     7 líneas   *  prerm                #!/bin/sh
 Package: alien
 Version: 8.05-2
 Section: alien
 Priority: extra
 Architecture: all
 Installed-Size: 251
 Maintainer: root <root@>
 Description: install non-native packages with dpkg
  Alien allows you to convert LSB, Red Hat, Stampede and Slackware
  Packages
  into Debian packages, which can be installed with dpkg.
[...]
```

ANOTACIONES

Para extraer únicamente los metadatos, tenemos la opción `-f` (`--field`):

```
$ dpkg -f alien_8.05-2_all.deb | less
Package: alien
Version: 8.05-2
Section: alien
Priority: extra
Architecture: all
Installed-Size: 251
Maintainer: root <root@>
Description: install non-native packages with dpkg
  Alien allows you to convert LSB, Red Hat, Stampede and Slackware
  Packages
  into Debian packages, which can be installed with dpkg.
```

Si necesitamos extraer los ficheros del paquete en un directorio que no sea el de instalación, podemos usar la opción `-x` (`--extract`). Si deseamos ver cada fichero según está siendo extraído ("verbose"), usaremos `-X` (`--vextract`):

```
$ mkdir /tmp/alien1
$ dpkg -x alien_8.05-2_all.deb /tmp/alien1/
./
./usr/
./usr/bin/
./usr/bin/alien
./usr/share/
./usr/share/alien/
./usr/share/alien/patches/
./usr/share/alien/patches/motif-devel_2.1.10-7.diff.gz
./usr/share/alien/patches/motif_2.1.10-7.diff.gz
[...]
```

Para extraer únicamente los datos de control del paquete sin necesidad de extraer todos los ficheros, podemos usar `-e` (`--control`):

```
$ mkdir /tmp/alien1/DEBIAN
$ dpkg -e alien_8.05-2_all.deb /tmp/alien1/DEBIAN
$ ls -l /tmp/alien1/DEBIAN
total 5
-rw-r--r--    1 root     root          518 nov 17 18:03 control
-rw-r--r--    1 root     root         1989 nov 17 18:03 md5sums
-rwxr-xr-x    1 root     root          249 nov 17 18:03 postinst
-rwxr-xr-x    1 root     root          192 nov 17 18:03 prerm
```

ANOTACIONES

Asimismo, con -b (--build) podemos construir un paquete .deb a partir de un directorio que contenga los datos y un directorio adicional DEBIAN con los metadatos y scripts de control.

Con esta opción se puede reconstruir un paquete previamente extraído con `-x` y `-e`:

```
$ dpkg -b /tmp/alien1
dpkg-deb: construyendo el paquete `alien' en `/tmp/alien1.deb'.
$ ls -l /tmp/alien1.deb
-rw-r--r--    1 root     root       112722 nov 17 18:03 /tmp/alien1.deb
```

Para crear un paquete .deb, lo único que hace falta es copiar en un directorio los ficheros que se quieran incluir en una estructura de directorios relativa al directorio raíz (como si fuese un *chroot*) y añadir un directorio DEBIAN en el que se incluirá un fichero de control con los datos relativos al paquete con las siguientes líneas:

`Package`: nombre del paquete.

`Version`: número de versión.

`Section`: sección temática en la que irá incluido (p. ej. en dselect).

`Architecture`: plataforma sobre la que funcionará el paquete.

`Depends`: lista de paquetes de los que depende. Cada paquete puede ir acompañado de un número de versión entre paréntesis precedido de "<<" (menor que), "<=" (menor o igual), "=" (exactamente la versión especificada), ">=" (mayor o igual), ">>" (sólo mayor que).

`Enhances`: especifica que la funcionalidad de determinado(s) paquete(s) será mejorada si se instala éste (por ejemplo, en caso de que sea un paquete de documentación).

`Pre-Depends`: especifica paquetes que deben estar ya instalados obligatoriamente antes de instalar este paquete (no permite instalaciones de "más de un paquete" en orden no definido).

ANOTACIONES

`Recommends`: especifica paquetes relacionados que comúnmente son instalados con este paquete.

`Suggests`: paquetes que tienen alguna relación con éste, pero no tienen por qué ser instalados para un funcionamiento normal.

`Installed-Size`: define el espacio que ocupará el paquete una vez instalado.

`Maintainer`: nombre y dirección de correo del encargado del paquete en formato "`Nombre <correo@dominio>`".

`Conflicts`: paquetes que no pueden estar instalados al mismo tiempo que éste para su correcto funcionamiento.

`Replaces`: este paquete sustituye a determinados otros paquetes (que no podrán estar instalados al mismo tiempo).

`Description`: un texto descriptivo del paquete.

Además de este fichero de control, se deben especificar dos *scripts* (shell) para ser ejecutados en el momento de la instalación (`postinst`) y de la desinstalación (`prerm`).

6.5.5. Alien (convertir paquetes DEB, RPM y TGZ)

Para paliar el problema de distribución que se presenta al haber diferentes formatos de paquetes entre las distintas distribuciones, se puede recurrir al programa alien que permitirá convertir paquetes ya creados entre los formatos rpm, deb, tgz y algunos más.

6.5.6. Comandos básicos Alien

La forma básica de uso es con las opciones `--to-deb` y `--to-rpm`:

```
$ alien --to-rpm alien_8.05_all.deb
alien-8.05-2.noarch.rpm generated
```

Igualmente, si deseamos pasar de deb a rpm:

```
$ alien --to-deb alien-8.05-2.noarch.rpm
alien_8.05-3_all.deb generated
```

También podemos instalar directamente el paquete convertido (p. ej. instalar un paquete `.rpm` en un sistema basado en `.deb`):

```
$ alien --to-deb -i alien-8.05-2.noarch.rpm
(Leyendo la base de datos ...
7257 ficheros y directorios instalados actualmente.)
Preparando para reemplazar alien 8.05 (usando alien_8.05-_all.deb) ...
Desempaquetando el reemplazo de alien ...
Configurando alien (8.05-3) ...
```

Para distribuir paquetes convertidos de esta forma, hay que tener en cuenta que alien, por defecto, aumenta en uno el número menor de versión (en el anterior ejemplo de 8.05-2 a 8.05-3). Esto se puede evitar especificando la opción `-k`:

```
$ alien --to-deb -k alien-8.05-2.noarch.rpm
alien_8.05-2_all.deb generated
```

Si lo que nos interesa es poder compilar el paquete en cuestión en la plataforma deseada pero pudiendo cambiar antes algo en él, tenemos la opción `-g` que extraerá el contenido en un directorio temporal:

```
$ alien -g alien-8.05-2.noarch.rpm
Directories alien-8.05 and alien-8.05.orig prepared.
```

En este caso, dado que alien por defecto exporta al formato `.deb`, se habrán creado dos directorios desde los que se podrá crear un paquete `.deb` y un paquete `.rpm` respectivamente.

6.6. Conclusiones

El contenedor `tar` guarda estructuras completas de archivos y directorios y junto con el compresor gzip es capaz de distribuir software

de forma homogénea pero inconsistente, es decir, su instalación es independiente de otros paquetes de software e incluso de una instalación idéntica anterior.

rpm y deb toman todo lo bueno de los anteriores guardando una consistencia de datos y versiones necesaria para la distribución del software por Internet y las instalaciones desatendidas.

La evolución de estos sistemas consistentes y su compatibilidad determinará el o los sistemas de distribución de software del futuro.

Lo más deseable: un *front-end* independiente del sistema de distribución que compatibilice versiones y unifique bases de datos, pudiendo gestionar de manera transparente cualquiera de ellos. La comunidad está en ello.

ANOTACIONES

7. Sistemas de creación de documentación

7.1. Introducción

Este capítulo cubre los distintos aspectos relacionados con los sistemas de documentación más aceptados dentro del mundo del software libre y de fuente abierta. La documentación en el ámbito de software juega un papel importantísimo como complemento indispensable de los programas informáticos. Se plantean las diferentes motivaciones que llevan a tener sistemas de documentación libre y se presentan los distintos tipos de licencias con los que la documentación es distribuida. También se analizan cada uno de los sistemas de documentación en forma particular que le permiten crear documentos básicos.

En particular se describe desde la simple documentación de una página de manual, pasando por documentación de desarrollo, documentación de breves manuales, hasta la posibilidad de escribir libros completos; todo con carácter profesional y útil para obtener cooperación, como se espera en el mundo del software libre.

7.2. Objetivos

Una vez estudiado este capítulo, el lector podrá hacer lo siguiente:

- Crear documentos básicos en cada uno de los sistemas planteados.
- Conocer cada uno de los sistemas de documentación, lo que le permitirá poder decidir qué sistema utilizar para documentar su software, de acuerdo a las necesidades del caso.
- Crear páginas de manual.
- Crear documentos cortos (HowTo, manuales, etc.).
- Crear libros de documentación (manuales, tratados, etc.).

ANOTACIONES

ANOTACIONES

7.3. Documentación libre: estándares y automatización

Un sistema o software pobremente documentado carece de valor aunque haya funcionado bien en alguna ocasión. En el caso de programas pequeños y poco importantes que sólo se utilizan durante un corto periodo de tiempo, unos cuantos comentarios en el código podrían ser suficientes. No obstante, la mayoría de los programas cuya única documentación es el código no tienen aceptación y es imposible mantenerlos. Dedicar un poco de esfuerzo a la documentación, incluso dentro de los límites de un pequeño proyecto, constituye una muy buena práctica.

Aprender a documentar software es una tarea complicada y exige un criterio de ingeniería maduro. Documentar escuetamente es un error habitual, pero el otro extremo puede resultar igual de perjudicial: si escribe documentaciones extensas, éstas atosigarán al lector y constituirán una carga a la hora de mantenerlas. Es esencial documentar sólo los asuntos correctos. La documentación no sirve de ayuda para nadie si su extensión desanima a la gente a la hora de leerla.

Este apartado provee al lector de las directrices sobre cómo documentar, principalmente orientado a sistemas de software libre, es decir, documentar con herramientas libres y en formatos libres. Asimismo, le proporcionará una estructura esquemática de tipos de documentos, quedando a su criterio la utilización de estos ejemplos.

7.3.1. La documentación del software

Una parte fundamental del software es la documentación que lo acompaña. La documentación es una pieza tan importante de un programa que en ocasiones éste será inútil sin ella.

La documentación del software está dividida en distintos tipos de acuerdo a lo que ella documente, y generalmente existe:

- Documentación de desarrollo:
  – Análisis, requisitos, especificaciones
  – Diagramas

– Comentarios en códigos
– Documentación de pruebas, etc.

• Documentación de programa:
– Ayuda en línea
– Página de manual (man)

• Documentación de usuario:
– Manual de uso
– Libros y tutoriales
– Guías de enseñanza o autoaprendizaje

Obviamente, ésta es una lista reducida de las posibilidades de documentación en torno a un software determinado y la documentación que se utilizará dependerá del alcance del software en cuestión.

Otra característica de la documentación del software es que la misma debe acompañar el desarrollo o evolución del software que documenta. Es decir, de la misma forma que el software avanza y se desarrolla, la documentación debe avanzar y desarrollarse conjuntamente, de manera que la última versión de la documentación refleje las características y el estado de la última versión del software.

7.3.2. El problema de documentar software libre

Si la documentación debe evolucionar junto al software, cuando hablamos de software libre nos encontramos ante el problema de que estamos documentando un software potencialmente realizado por muchas personas, que puede ser libremente modificado y mejorado. Es obvio, entonces, que la documentación del software libre debe poder ser libremente modificada y mejorada para acompañar la evolución del software libre. De hecho, se considera que software libre sin documentación libre no es un software libre, ya que el desarrollo libre del software se puede ver perjudicado por la falta de un acceso libre a su documentación.

Pensad, por ejemplo, el caso en que tomarais un software libre y le realizarais algunas mejoras; si la documentación no fuese libre, entonces no podría reflejar su mejora en el mismo documento, lo que significaría un problema importante para el desarrollo o al menos para la difusión y aplicación de su modificación.

ANOTACIONES

Contenido complementario

El software libre es aquel que permite al usuario copiarlo, mejorarlo y distribuirlo libremente.

El software privativo es aquel que carece de alguna o de la mayoría de las libertades que se reciben con el software libre.

Nota

http://www.gnu.org/copyleft/fdl.html

http://creative-commons.org/text/

7.3.3. Licencias libres para la documentación

Todo software, ya sea libre o privativo, está acompañado por un contrato de licencia.

La documentación que acompaña al software resulta, al igual que éste, en una producción del intelecto humano, por lo cual le son aplicables las leyes de derechos de autor o de copyright.

Por este motivo, para poder copiar, modificar o distribuir la documentación, es necesario tener el permiso del autor de dicha documentación o tener un documento que le otorga estos permisos. Este permiso se corporiza en una licencia para la documentación.

La licencia para la documentación de software libre generalmente es la misma que la del software al que acompaña, aunque existen algunos documentos publicados bajo licencias específicas para documentación como la *free documentation licence* (FDL) de la Free Software Foundation, aunque últimamente mucha documentación está siendo entregada con la licencia conocida como *creative commons*.

En todos los casos en los que vayáis a realizar modificaciones del trabajo intelectual ajeno, deberéis conocer si tenéis permiso para hacerlo, si está especificado en algún lugar y, en caso de tener una licencia, deberéis leerla.

7.3.4. Formatos libres y propietarios

Tan importante como tener las libertades para la documentación, es documentar en formatos libres, de manera que, una vez que su documento llegue a los lectores, pueda ser accedido, modificado y vuelto a distribuir con todas las libertades, sin depender de restricciones impuestas al software de acceso o modificación. Esto se logra documentando en formatos libres.

Imaginemos por un momento que alguien documenta su software con una herramienta que no se puede copiar libremente, donde cada copia requiere de una licencia. Podría hacer llegar sus documentos a distintas personas, pero éstas estarían obligadas a aceptar un contrato de licenciamiento para que puedan leer sus documentos.

Por otro lado, considerad que guarda sus documentos en un formato (formato binario, formato de archivo de datos) que sólo conoce y maneja una empresa y que accede y modifica mediante el software de esta misma empresa. Pero a los pocos años esa empresa decide modificar sus formatos de archivos, o la empresa cierra el proyecto, o la empresa abandona el mercado. Su necesidad de acceder a la información allí guardada seguirá existiendo y tal vez no pueda acceder a sus propios datos o textos.

Empresarial o estatalmente, manejar archivos de diez años de antigüedad (contabilidad, declaraciones juradas, etc.) es normal y en ocasiones obligatorio. Posiblemente, en diez años, ni siquiera el soporte físico (disquetes, CD-ROM, DVD, cintas) sea el mismo, mucho menos el software para acceder a los datos, por lo cual, solamente si se han guardado los datos en formatos estándares que son conocidos por todos y accesibles para muchos sistemas, podremos visualizarlos.

Los formatos libres más usados. txt, rtf, html, TeX, XML, etc. están avalados por organizaciones que establecen los estándares para cada uno de ellos.

7.3.5. Herramientas de control y administración de versiones

Como indicábamos anteriormente, la documentación debe evolucionar en concordancia con el software que documenta. Es decir, a cada nueva versión del software le corresponderá una nueva versión de la documentación. Al menos, la documentación deberá indicar a qué versiones del software le son aplicables las distintas opciones documentadas.

El software libre se crea y desarrolla generalmente por grupos de trabajo guiados por un líder de proyecto. La documentación también se realiza mediante un proceso semejante. Para esto, se utilizan distintas herramientas que permiten controlar y versionar la documentación en forma cooperativa y automática, siendo la más utilizada el sistema CVS (*concurrent versions system*), que consiste en un sistema donde cada autor puede subir sus textos o modificaciones, y el sistema lleva a cabo un control automático de los cambios entre versiones.

ANOTACIONES

El sistema CVS no es un sistema específico para documentación, también es utilizado para código fuente o cualquier sistema de textos.

Con la extensión de Internet, en los últimos tiempos se han desarrollado algunos sistemas de documentación cooperativa en línea que permiten un trabajo eficaz en grupos de autores que trabajan simultáneamente. En particular abordaremos más adelante el sistema WikiWikiWeb, aunque se están estudiando otros sistemas para este mismo fin.

7.4. Creación de páginas de manual

Las aplicaciones, utilidades y los comandos usualmente tienen sus páginas de manual correspondientes (llamadas *man pages*), que muestran las opciones disponibles y valores de archivos o ejecutables. Las páginas man están estructuradas de forma que los usuarios puedan buscar fácilmente la información pertinente, lo que es muy importante cuando se está trabajando con comandos con los que nunca se ha trabajado antes.

Se puede acceder a las páginas man a través del shell escribiendo el comando **man** y el nombre del ejecutable.

7.4.1. Secciones de las páginas de manual

Cada página de manual está dividida en secciones que varían de acuerdo a la complejidad del comando o aplicación que documentan, la siguiente es una lista de las secciones estándares:

NAME (nombre): muestra el nombre del ejecutable y una breve explicación de la función que realiza.

SYNOPSIS (sinopsis): muestra el uso común del ejecutable, tal como cuáles opciones son declaradas y qué tipo de entradas (archivos, valores, argumentos) soporta el ejecutable.

DESCRIPTION (descripción): muestra las opciones y valores asociados a un archivo o ejecutable.

ANOTACIONES

SEE ALSO (véase también): muestra otros términos, archivos o programas relacionados con el ejecutable de la página man consultada.

Otras secciones ya dependen directamente del comando documentado en la página man, como por ejemplo:

OPTIONS (opciones): muestra una descripción de cada uno de los parámetros que puede recibir el ejecutable.

EXAMPLES (ejemplos): muestra una lista de invocaciones del ejecutable más usuales.

FILES (ficheros): lista de archivos involucrados para la debida ejecución del programa, generalmente archivos de configuración.

AUTHORS (autores): autores del comando o programa y sus direcciones de correo electrónico.

BUGS (fallos): muestra una lista de fallos conocidos en el ejecutable o el programa.

HISTORY (historia): muestra una lista de las versiones del documento man y de sus modificaciones.

Obviamente, el autor del man dispone de flexibilidad para agregar las secciones que entienda oportunas a los efectos de tener debidamente documentado su comando o programa; de la misma manera, no todas las páginas de manual poseen todas las secciones anteriores.

7.4.2. Camino de búsqueda de páginas man

Cuando es invocado el comando man para mostrar la página de manual de un comando determinado, es necesario conocer dónde se encuentran disponibles los archivos que contienen los textos que hay que desplegar, esto se logra mediante la utilización de MANPATH.

MANPATH permite determinar la ruta para la búsqueda de las páginas de manual después de la invocación del comando man. Generalmente,

una distribución de GNU/Linux ya tiene una configuración apropiada de MANPATH y no es necesario modificarla.

MANPATH tiene dos formas de ser configurado. Una es mediante la configuración de la variable de entorno $MANPATH, que se realiza generalmente en el archivo `/etc/profile` del sistema con la línea `MANPATH=/usr/man:/usr/share/man:/usr/X11R6/man`. Y otra es mediante el archivo `/etc/manpath.conf`.

La sintaxis para definir MANPATH es semejante a la definición del PATH para búsqueda de ejecutables en el sistema: la ruta, delimitada por "dos puntos". MANPATH tiene que contener la ruta en la base de las jerarquías de capítulos (ver la siguiente sección) de las páginas man.

7.4.3. Jerarquía de capítulos de las páginas man

El conjunto de las páginas man se clasifica en capítulos que están destinados a optimizar las búsquedas y a permitir que los sistemas sólo tengan instalados aquellos capítulos que necesitan, dependiendo de su finalidad.

Bajo el directorio definido en MANPATH, se encuentra una jerarquía de directorios definidos como **manX**, donde **X** corresponde al capítulo de la página man. Los capítulos son los siguientes:

1) Comandos del usuario

2) Llamadas al sistema (*system calls*)

3) Librerías y funciones del lenguaje C

4) Dispositivos

5) Formatos de archivos

6) Juegos

7) Varios

8) Comandos de administración del sistema

ANOTACIONES

7.4.4. Generación de páginas man usando herramientas estándares

Las páginas de man son escritas en un lenguaje de marcas, generalmente conocido como `archivo nroff`; de hecho, existen otros procesadores para estos archivos de marcas (como `troff` o `groff`), pero todos soportan una sintaxis semejante. Nroff es un lenguaje de marcas más primitivo que SGML o HTML. Por otra parte, es un lenguaje de macros, por lo que podemos realizar con él tareas complejas.

Para escribir su propia página man, primeramente tiene que tener un archivo fuente `nroff`, por ejemplo:

```
.TH "miprograma" 1
.SH NAME
miprograma \- Mi primer programa
.SH SYNOPSIS
.B miprograma
miprograma [-laFh]
.SH DESCRIPTION
Mi primer programa para probar páginas man.
.SH OPTIONS
.TP
.B \-l
Parámetro l de miprograma.
.TP
.B \-a
Parámetro a de mi programa.
.TP
.B otros parámetros . . .
.SH "SEE ALSO"
miotroprograma(1)
.SH BUGS
No hay errores reportados.
```

La macro .TH junto al nombre del programa tiene un 1. Esto significa que el programa que se está documentando se archivará en el capítulo 1 (`man1`) de las jerarquías de páginas man.

Una vez que se dispone del archivo nroff fuente, se necesita que el programa nroff lo interprete para producir el archivo de página man, con la siguiente línea:

```
nroff -man miprograma.nroff > miprograma.1
```

El resultado de la ejecución de este parámetro es dirigido a un archivo que se llama como el comando y que tiene la extensión del

ANOTACIONES

número del capítulo donde será archivado. Se puede ver el resultado con el comando:

```
less miprograma.1
```

Para que la página man quede a disposición para leer con el comando man, la debe colocar en el directorio correspondiente del capítulo.

7.4.5. Generación de páginas de manual usando `perlpod`

Nota

http://www.perl.org/about.html

Otra forma de realizar páginas de manual es utilizando el lenguaje de interpretación de *scripts* Perl mediante una utilidad llamada `pod2man` (además de pod2html, pod2tex y pod2text), que permite crear páginas de manual fácilmente. **pod** quiere decir “Plain Old Documentation”. Por ejemplo, para crear la página de manual de un programa dict2docbook, se puede escribir lo siguiente en un fichero miprograma.pod:

```
=head1 NOMBRE
miprograma - Mi primer programa
=head1 SYNOPSIS
S<miprograma [-laFh]>
=head1 DESCRIPCION
Mi primer programa para probar páginas man.
=head1 AUTOR
Juan Pueblo E<lt>jpueblo@mail.orgE<gt>.
=cut
```

Después se ejecuta

```
pod2man -lax miprograma.pod > miprograma.1
```

para crear la página de manual. La opción lax se usa para que se acepten nombres diferentes a los habituales para las secciones, pues en este caso están traducidos al español. La página creada puede ser leída con:

```
man -l miprograma.1
```

La sintaxis completa del pod está explicada en `man perlpod`, y es bastante simple; también se debe consultar la página de manual de pod2man. El único paquete necesario es perl.

Si el programa que se quiere documentar está escrito en perl, el código pod puede estar dentro del propio programa y puede servir tanto para documentar el código fuente, como para crear la documentación adicional para man.

7.5. TeX y LaTeX

TeX es un programa de Donald E. Knuth, que está orientado a la composición e impresión de textos y fórmulas matemáticas.

LaTeX es un paquete de macros que permite al autor de un texto componer e imprimir un documento con calidad, empleando patrones definidos. Originalmente, LaTeX fue escrito por Leslie Lamport y utiliza TeX como su elemento de composición. LaTeX es una potente herramienta de procesamiento de textos científicos que aún no ha sido sustituida por los modernos editores de texto en el mundo de las editoriales científicas y académicas. Hay quienes afirman que LaTeX es, probablemente, el mejor procesador de textos del mundo, pero tiene el inconveniente de que no se trabaja de forma *WYSIWYG (What you see is what you get)* lo que dificulta en gran medida su uso por parte de personas acostumbradas a editores de texto.

La gran problemática que presentan los editores de textos *WYSIWYG* es que el autor (usuario) debe estar continuamente observando el formato de los textos (el título va centrado, la numeración de las viñetas, el formato de los párrafos, etc.), mientras que con LaTeX el autor puede dedicarse exclusivamente al texto (contenido) sin necesidad de distraer la atención en la presentación. Esta separación entre contenido y presentación, en ambientes de producción (como un periódico) es natural y existen roles diferentes para cada una: el autor y el diseñador. Los editores de texto modernos obligan al autor a ocuparse de las tareas del diseñador.

Con LaTeX el autor solamente deberá ocuparse del texto, del contenido; dejará que el programa se encargue de la presentación. Así, un mismo texto será presentado de diferente modo si es para imprimir en formato A4 o carta, si su destino es un pdf o una página web.

ANOTACIONES

LaTeX no sólo se ocupa de las tareas de presentación final, también de la numeración de los capítulos, los índices temáticos, los enlaces cruzados, la organización de la bibliografía citada, los enlaces web, la inclusión de otros archivos, la numeración de páginas, las notas al pie, etc.

LaTeX nació en UNIX, con lo que está disponible en plataformas con este sistema operativo, incluyendo ordenadores personales con GNU/Linux. Como ocurre con muchos programas de libre distribución, ha sido importado con éxito a otras muchas plataformas, incluyendo DOS/Windows. Éste es otro punto a favor de LaTeX: la posibilidad de trabajar con él virtualmente en todas las plataformas hardware y software existentes.

7.5.1. Instalación

Sobre la instalación de LaTeX no vamos a detallar demasiado, ya que afortunadamente forma parte de los paquetes de las distribuciones de GNU/Linux más usadas actualmente, por lo que la instalación es completamente automática. Destacamos, sin embargo, que hay que tratar de instalarse todos los paquetes de LaTeX (excepto algunos evidentes, como los soportes de idiomas orientales y otras rarezas), haciendo énfasis en las herramientas de visualización y conversión de ficheros procesados .DVI (dvips, xdvi...), lo que nos permitirá imprimir o ver el resultado por pantalla.

7.5.2. Utilización

El uso de LaTeX es parecido al de un compilador, por eso es denominado "procesador de textos" (a diferencia de los "editores de texto"). Los pasos que hay que seguir en un documento son los siguientes:

1) Escribir el texto *código fuente* con la sintaxis (o marcas) de LaTeX con cualquier editor de textos ASCII (se recomienda alguno que destaque las marcas con colores).

2) Se *compila* el fichero escrito, escribiendo el comando `latex fichero.tex`.

ANOTACIONES

3) Se visualiza el resultado en formato gráfico con el visor de dvi con el comando `xdvi fichero.dvi.`

4) Se reedita el fichero para corregir o continuar el trabajo, y finalmente se imprime con el comando dvips fichero.dvi. o se transforma a formatos pdf, html o rtf.

7.5.3. Fichero fuente LaTeX

El siguiente es un ejemplo de un documento básico en latex:

```
\documentclass[a4paper,11pt]{article}
\userpackage[activeacute,spanish]{babel}
\author{Juan Pueblo}
\title{Mi primer documento LaTeX}
\begin{document}
\maketitle
\tableofcontents
\section{Introducción}
La primer parte del documento y un párrafo.
```

Otro párrafo de la introducción:

```
\section{Ultima parte}
```

La última parte y aquí se acaba:

```
\end{document}
```

7.5.4. Clases de documentos

La instalación estándar de Latex ya trae un grupo importante de documentos que pueden ser utilizados para lograr distintos resultados:

- **article** para artículos de revistas especializadas, ponencias, trabajos de prácticas de formación, trabajos de seminarios, informes pequeños, solicitudes, dictámenes, descripciones de programas, invitaciones y muchos otros.
- **report** para informes mayores que constan de más de un capítulo, proyectos de fin de carrera, tesis doctorales, libros pequeños, disertaciones, guiones o similares.

ANOTACIONES

- **book** para libros.
- **slide** para transparencias. Esta clase de documentos producirá páginas con tipos grandes de caracteres.

7.5.5. Extensiones o paquetes

Una de las ventajas de LaTeX es que puede ser extendido para abarcar situaciones donde el sistema básico no soluciona el problema, como son inclusión de gráficos, textos a colores, códigos fuente a partir de ficheros, etc. Para esto se utilizan los paquetes mediante la orden:

```
\usepackage[opciones]{paquete}
```

En el ejemplo básico previo se incluye el paquete *babel*, que permite la expansión de los caracteres para incluir los del idioma español.

7.5.6. Edición WYSWYG con LaTeX

Nota

http://www.lyx.org

Si bién LaTeX está orientado a trabajar como se indica anteriormente, existe un editor WYSWYG llamado **Lyx** que hace un procesamiento en tiempo de escritura del documento LaTeX. No obstante, debido a que el resultado de un texto procesado con LaTeX es fijo, de acuerdo a la clase del documento, el lyx no puede ser usado en la misma forma de flexibilidad que un editor WYSWYG; aunque es muy fácil acostumbrarse a escribir con LYX. Lyx también está portado para correr en múltiples sistemas operativos.

Algunos editores WYSWYG poseen la capacidad de guardar los documentos en formato de fuente LaTeX; también hay herramientas que convierten documentos de otros formatos a .tex. No obstante, para el usuario medianamente entrenado en LaTeX, siempre va a ser más fácil editar con cualquier editor y procesar con latex sus textos.

7.6. SGML

SGML son las siglas de *standard generalized markup language* y es un sistema para organizar y etiquetar elementos en un documento.

ANOTACIONES

SGML fue desarrollado como un estándar internacional por la International Organization for Standards (ISO) in 1986. SGML por sí mismo no especifica ningún formato en particular, pues es la norma que determina las reglas para incluir etiquetas de marcado en los documentos. Y estas etiquetas pueden ser interpretadas por elementos de lectura, presentación, recuperación de información, etc., de diferentes maneras.

SGML es utilizado en grandes documentos que son objeto de frecuentes revisiones y requieren ser impresos en diferentes formatos. Como SGML es un sistema complejo y amplio, no es utilizado en forma masiva. No obstante, el estándar **HTML** para el World Wide Web es una implementación de SGML; es decir, el HTML es un conjunto de etiquetas que siguen el estándar SGML al ser incluidas en un documento.

7.6.1. Documentación en formato html

Son muchos los editores que permiten guardar documentos en formato HTML, dejándolos listos para ser publicados en un servidor web. No obstante, cuando hablamos de documentación de software libre, tenemos que pensar en documentos que deben ser modificados por distintas personas y frecuentemente, por lo que se requieren herramientas más complejas que un editor de textos.

Herramienta web de documentación cooperativa: el WikiWikiWeb

Frente a la problemática de editar documentación por diferentes autores, que se mantengan versiones de cada documento y que esos documentos estén en línea inmediatamente, se creó una herramienta llamada wikiwikiweb.

El término de WikiWiki (*wiki wiki* significa *rápido* en la lengua hawaiana) se utiliza en muchos sitios web Internet para nombrar una colección de páginas web de hipertexto, cada una de las cuales puede ser visitada y editada por cualquier persona. Una versión web de un wiki también se llama WikiWikiWeb. Se trata de un simple juego de palabras, ya que las iniciales son **WWW** como en la World Wide Web.

ANOTACIONES

La forma abreviada wiki denomina a la aplicación de informática colaborativa que permite crear colectivamente documentos web usando un simple esquema de etiquetas y marcas, sin que la revisión del contenido sea necesaria antes de su aceptación para ser publicado en el sitio web en Internet.

Dada la gran rapidez con la que se actualizan los contenidos, la palabra *wiki* adopta todo su sentido. El documento de hipertexto resultante, denominado también *wiki* o *WikiWikiWeb*, lo produce típicamente una comunidad de usuarios. Muchos de estos lugares son inmediatamente identificables por su particular uso de palabras en mayúsculas, o texto capitalizado; uso que consiste en poner en mayúsculas las iniciales de las palabras de una frase y eliminar los espacios entre ellas, como por ejemplo en EsteEsUnEjemplo. Esto convierte automáticamente a la frase en un enlace. Este wiki, en sus orígenes, se comportaba de esa manera, pero actualmente se respetan los espacios y sólo hace falta encerrar el título del enlace entre dos corchetes.

El objetivo de un wiki es democratizar la creación y el mantenimiento de las páginas, al eliminar el *síndrome de un único webmaster o administrador*.

El gran potencial del wiki radica en que no es necesario aprender a utilizar complicadas etiquetas HTML para escribir de forma sencilla documentos y establecer enlaces desde el sitio web.

Nota

http://www.c2.com/cgi/wiki? WelcomeVisitors

El primer WikiWikiWeb fue creado por Ward Cunningham, quien inventó y dio nombre al concepto wiki, y produjo la primera implementación de un servidor WikiWiki. En palabras del propio Ward, un wiki es "la base de datos en línea más simple que posiblemente pudiera funcionar" (*the simplest online database that could possibly work*).

En principio, un wiki se usa para cualquier cosa que sus usuarios deseen y está siendo adoptado para documentar mucho software libre en forma cooperativa. El formato se presta a la colaboración, colaboración que involucra a cualquier persona.

Nota

http://es.wikipedia.org

Un ejemplo es Wikipedia; un wiki que tiene como misión específica ser una enciclopedia libre y actualizada por todo aquel que lo desee.

ANOTACIONES

7.6.2. Documentación en formato Docbook

DocBook es un dialecto de SGML (como lo es HTML) especialmente orientado a la escritura de documentación técnica. En particular es una aplicación del estándar SGML/XML, que incluye una DTD propia y que se utiliza de manera más destacada en el área de la documentación técnica, especialmente para documentar todo tipo de material y programas informáticos. Existe un Comité Técnico de DocBook en OASIS (originalmente SGML Open) que mantiene y actualiza este estándar. DocBook inicialmente comenzó como una DTD de SGML, pero a partir de la versión 4 existe un equivalente para XML.

Nota

http://www.oasis-open.org/

DocBook es muy utilizado en algunos contextos, entre los que destacan Linux Documentation Project (Proyecto de documentación Linux), las referencias de las API de GNOME y GTK+, así como la documentación del núcleo Linux. Las páginas man del entorno operativo Solaris se generan también a partir de documentos que utilizan las DTD de DocBook.

Norman Walsh y el equipo de desarrollo del DocBook Open Repository mantienen un conjunto de hojas de estilo DSSSL y XSL para generar versiones PDF y HTML de documentos DocBook (así como para desarrollar otros formatos, incluyendo páginas de referencia man y de ayuda en HTML). Walsh es también el principal autor del libro *DocBook: The Definitive Guide*, la documentación oficial de DocBook. Este libro se puede obtener bajo licencia GFDL o en su versión impresa está editada por O'Reilly & Associates."

Instalación y consideraciones preliminares

Para poder utilizar DocBook y poder sacar un mejor provecho de la escritura, se recomienda tener instalados las siguientes utilidades:

DocBook y utilidades: estos paquetes son requeridos para que todo funcione debidamente; generalmente las distribuciones de GNU/Linux ya traen paquetes prontos para instalar.

emacs: si bien los documentos pueden ser editados con cualquier editor de textos, Emacs es el único editor orientado al contexto (funcionalidad muy útil a la hora de editar DocBook).

ANOTACIONES

Modo PSGML de emacs: soporte para trabajar con la DTD de DocBook en el momento de edición en Emacs.

Fichero fuente DocBook

El siguiente es un ejemplo de un documento básico DocBook:

```
<!doctype book PUBLIC "-//OASIS//DTD DocBook V4.1//EN">
 <book lang=es>
   <bookinfo>
     <title>Mi primer documento DocBook</title>
     <subtitle>Un subtítulo</subtitle>
   </bookinfo>
   <toc></toc>
   <!-- Puede poner un comentario o directamente comenzar a escri-
        bir el documento -->
   <chapter>
     <title>Introducción</title>
     <para>La primera parte del documento y un párrafo.</para>
     <para>Otro párrafo de la introducción</para>
   </chapter>
   <chapter>
     <title>Última parte</title>
     <para>La última parte y aquí se acaba.</para>
   </chapter>
</book>
```

La utilización para editar este documento de una herramienta orientada al contexto (emacs+psgml) le permitirán tener una sintaxis controlada, es decir, no le dejarán escribir <title> si aún no ha abierto un <chapter>, ya que el editor sabe qué etiquetas tiene disponibles dentro del contexto en el que se encuentra escribiendo.

Utilización

Una vez que tenga su documento escrito en DocBook, debería revisarlo para conocer si no ha cometido ningún error de sintaxis (cerrar una etiqueta; texto en una etiqueta, etc.), para esto existe el siguiente comando:

```
nsgmls -s midocumento.sgml
```

Ahora deberá decidir qué salida desea para su documento y lo puede hacer utilizando los procesadores de textos **db2html**, **db2ps**, **db2pdf**, **db2rtf**, de la siguiente forma:

db2pdf midocumento.sgml

ANOTACIONES

7.7. Doxygen. Documentación de código fuente

Doxygen es un sistema de documentación para códigos fuente de programas escritos en una variedad importante de lenguajes, entre los que se encuentra: C++, C, Java, Objective-C, IDL (Corba), PHP, C# y D.

Doxygen puede generar documentación para ser publicada en Internet o para ser procesada con LaTeX. También puede generar salidas en formatos rtf, postscript, pdf y páginas de manual de UNIX.

La idea detrás de Doxygen es documentar en el propio código fuente, a medida que éste se escribe, de forma tal que Doxygen extrae la documentación del propio código fuente.

Doxygen fue creado bajo GNU/Linux y Mac OS X, pero ha sido portado para la mayoría de los sistemas UNIX y también para Windows.

7.7.1. Utilización

Doxygen utiliza un archivo de configuración para determinar cómo debe procesar la documentación. Cada proyecto deberá tener su propio archivo de configuración que entre otras cosas incluirá qué código fuente debe ser analizado, qué directorios, etc.

Existe una forma simple de crear una plantilla del archivo de configuración con el comando:

```
doxygen -g archivo-config
```

Las más nuevas versiones de doxygen traen una utilidad llamado **doxywizard**, que permite editar el archivo de configuración de una forma gráfica mediante diálogos.

Para un proyecto pequeño constituido por una fuente en C o C++, no es necesario hacer modificaciones, ya que doxygen buscará los archivos de fuentes en el directorio actual.

Para generar la documentación basta con ejecutar el comando:

```
doxygen archivo-config
```

ANOTACIONES

7.7.2. Documentando en los códigos fuente

La documentación dentro de los códigos fuente debe ser realizada dentro de bloques especiales de texto que puedan ser reconocidos por doxygen y a su vez no interfieran con el compilador del lenguaje.

Dentro de cada bloque de documentación existen dos tipos de descripciones, que juntándolas se crea la documentación:

- descripción abreviada
- descripción detallada

Para marcar un bloque como una descripción detallada, como por ejemplo, utilizando el estilo JavaDoc, que es un formato de bloque de comentario tipo C, iniciando con un asterisco, como este ejemplo:

```
/**
* ... texto ...
*/
```

También es posible utilizar un formato de comentario tipo Qt, que es un formato de bloque de comentario tipo C, iniciado con un símbolo de exclamación, como por ejemplo:

```
/*!
* ... texto ...
*/
```

En ambos casos, el asterisco intermedio es opcional, o sea, que se puede aceptar:

```
/*!
... texto ...
*/
```

Una tercera forma de marcar un bloque es con la notación de comentario de C++, que se individualiza con una barra adicional:

```
///
/// ... texto ...
///
```

ANOTACIONES

o también

```
//!
//! ... texto ...
//!
```

Para individualizar la descripción abreviada, también existen varias posibilidades de marcado.

Una forma es iniciar la línea con el comando

```
\brief
```

en alguno de los formatos de bloques detallados anteriormente, seguidos de una línea en blanco:

```
/*! \brief Descripción abreviada.
 * Continuación de la descripción abreviada.
 *
 * ... texto ...
 */
```

Si el archivo de configuración se configura con la opción JAVADOC_AUTOBRIEF en YES, es posible utilizar el estilo de bloques de JavaDoc que automáticamente inicia la descripción abreviada hasta el próximo punto seguido de un espacio o una nueva línea; como en este ejemplo:

```
/** Descripción abreviada que termina en un punto. Descripción
* detallada que comienza después.
*/
```

Un tercer método para diferenciar los tipos de descripciones es utilizar un comentario en formato C++ en una sola línea.

```
/// Descripción abreviada.
/** Descripción detallada */
```

o también dejando una línea en blanco entre una y otra descripción:

```
//! Descripción abreviada.

//! Descripción detallada
//! en más de un renglón
```

ANOTACIONES

En este último caso tendrá que tener explícitamente indicado JAVADOC_AUTOBRIEF en NO para que sean propiamente detectadas las descripciones.

Esto muestra como doxygen es flexible para adaptarse a los formatos de comentarios de cada uno de los desarrolladores de software.

7.8. Conclusiones

La documentación en software libre adquiere una relevancia mucho mayor que en sistemas de licenciamiento privativos, ya que en software libre se busca la cooperación en el desarrollo; por ello, tener una buena documentación ayudará a los desarrolladores y usuarios a conocer mejor y más rápidamente la herramienta de software, quedando habilitados a cooperar con el proyecto, ya sea informando de errores, proponiendo mejoras, aportando código o contestando preguntas a otros usuarios y desarrolladores.

Los sistemas de documentación aquí presentados permiten tener una idea de las características generales de cada uno de ellos y son una breve guía para poder construir un documento básico utilizando las herramientas descritas; dependerá ahora del alumno profundizar en el o los sistemas que considere más apropiados para obtener un conocimiento completo de ellos.

7.9. Otras fuentes de referencia e información

Referencias de Tex y Latex
Michael Doob. *A Gentle Introduction to TeX.*
George Gratzer. *First Steps in LaTeX.*

SGML
SGML for Dummies (SGML para principiantes)
Developing SGML DTDs: From text to Model to Markup

DocBook
DocBook: the definitive guide

Doxygen
http://www.stack.nl/~dimitri/doxygen/manual.html

ANOTACIONES

8. Comunidades virtuales y recursos existentes

8.1. Introducción

Desde sus comienzos, el desarrollo del software libre ha estado muy relacionado con el desarrollo de Internet.

En un principio los desarrolladores se comunicaban mediante listas de correo en las redes precursoras de Internet y en la medida en que se desarrollaba éstas, también lo hacia su manera de comunicarse.

Con el actual crecimiento de la red y los nuevos lenguajes de desarrollo web, hemos llegado a un punto en el que la cantidad de servicios ofrecidos en portales para los desarrolladores e ingenieros de software es enorme, y no sólo abarca herramientas que ayudan a éstos en su desarrollo diario, sino que ponen en contacto a patrocinadores de proyectos con desarrolladores, de modo que el software libre pueda crecer y expandirse.

Por otra parte, los desarrolladores encuentran lugares en los que pueden publicitarse ellos mismos, consiguiendo así una notoriedad muy valiosa para su desarrollo personal y laboral.

8.2. Objetivos

Tras concluir el capítulo, el lector será capaz de lo siguiente:

- Valorar las opciones en las que publicitar sus proyectos de software o los que son gestionados para terceros.

- Conocer el funcionamiento de los portales más comunes como Freashmeat y Sourceforge, aprendiendo el sistema de creación de

ANOTACIONES

una cuenta personal, una de proyecto y la gestión de ésta, no sólo del proyecto sino de sus subscriptores.

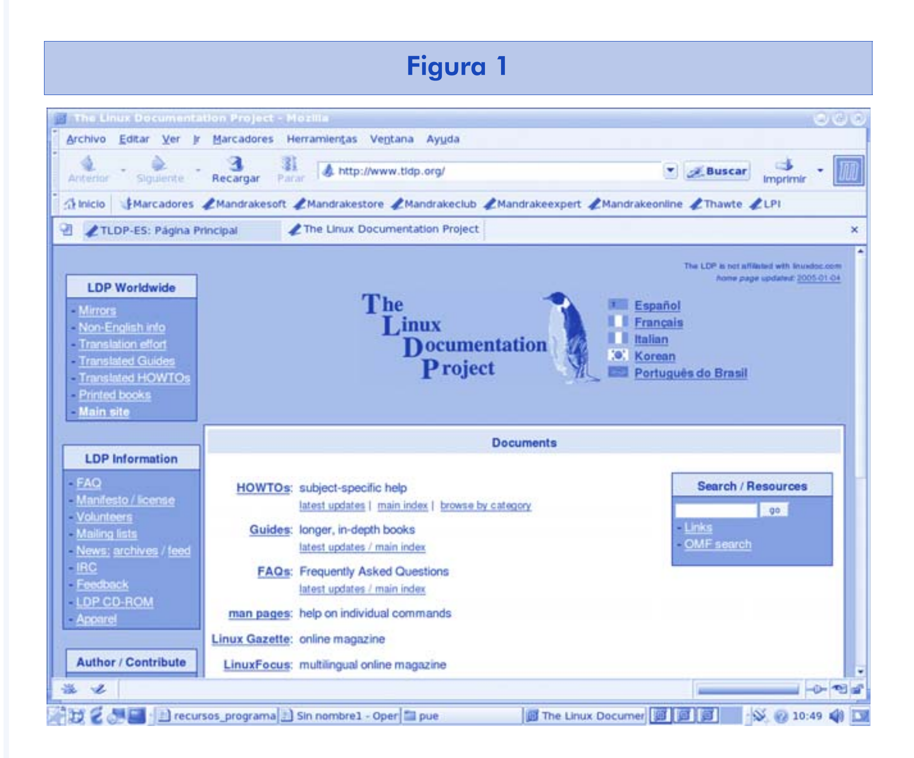
8.3. Recursos documentales para el ingeniero en software libre

En este punto tratamos de mostrar los recursos documentales más conocidos a través de los que el ingeniero de software libre dispondrá no sólo de manuales, sino de cursos, guías, normativa, tipos de licencia e incluso en algunos casos proyectos completos de los que tomar ejemplo para elaborar el nuestro.

8.3.1. http://www.tldp.org

Las siglas tldp significan The Linux Documentation Proyect. Esta página tiene versión en distintos idiomas:

Figura 1



Desde éste podemos acceder a mucha documentación libre:

1) Guías
2) Cómos
3) Faqs

ANOTACIONES

4) Páginas de manual Linux
5) Manuales
6) Cursos

Además, tendremos acceso a varios servicios como el de publicaciones, de desarrollo de proyectos, enlaces con otros proyectos y páginas en general

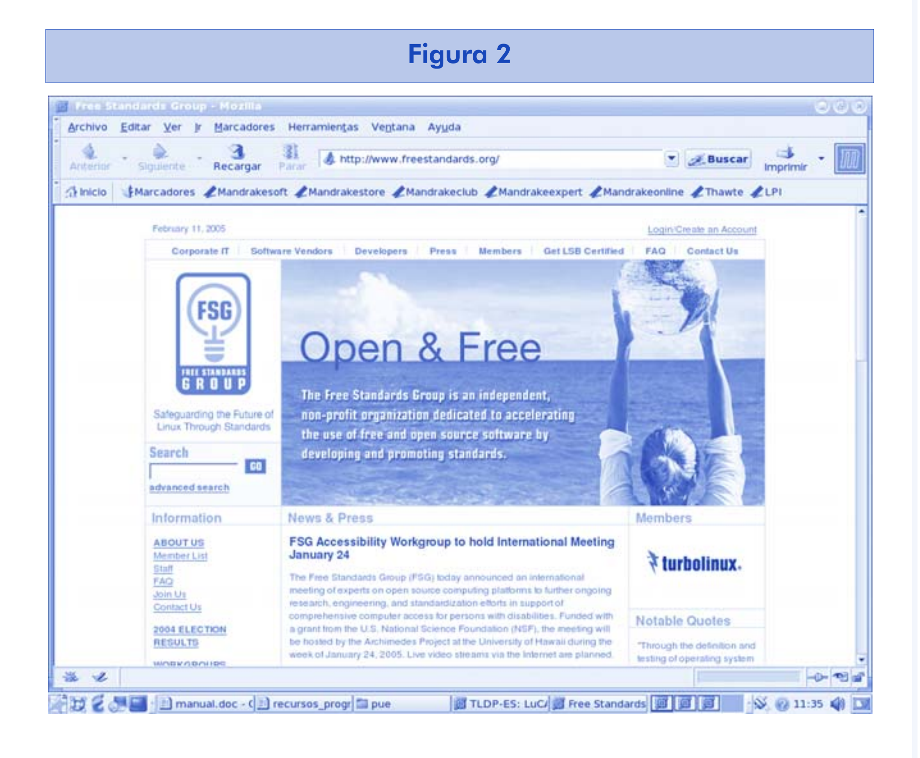
8.3.2. http://www.freestandards.org

Esta organización (Free Standards Group) independiente tiene como intención acelerar el uso del software libre y de código abierto mediante el desarrollo y la promoción de estándares o reglas.

Para ello, crea herramientas de ayuda para desarrolladores de software libre. Además, se encarga de testear y certificar mediante programas que el software desarrollado sigue los estándares existentes.

Otra función de este grupo es ayudar a migrar al sistema operativo Linux para aquellos que estén acostumbrados a manejar otros entornos.

Figura 2



En la página tendremos información mediante noticias y prensa acerca de todo el desarrollo de software libre, acerca de empresas

relacionadas con el campo y sus nuevos proyectos dentro del área de Linux.

Nota

http://www.linuxbase.org

Se dan certificaciones a vendedores de software a través de LSB (Linux Standard Base). Desde la página hay un enlace a LSB a través del cual se pueden recibir las certificaciones antes descritas. LSB es un grupo de trabajo de *freestandards*, pero no el único.

Además, están los siguientes:

- Openi18n. http://www.openi18n.org/
- LANANA. http://www.lanana.org/
- Openprinting. http://www.openprinting.org/
- Accesibility. http://accessibility.freestandards.org
- DWARF. http://dwarf.freestandards.org/
- Open Cluster. http://www.opencf.org/home.html

Cada uno de los grupos de trabajo se encarga de un área distinta dentro del entorno del software libre y por tanto de Linux.

8.3.3. http://www.unix.org

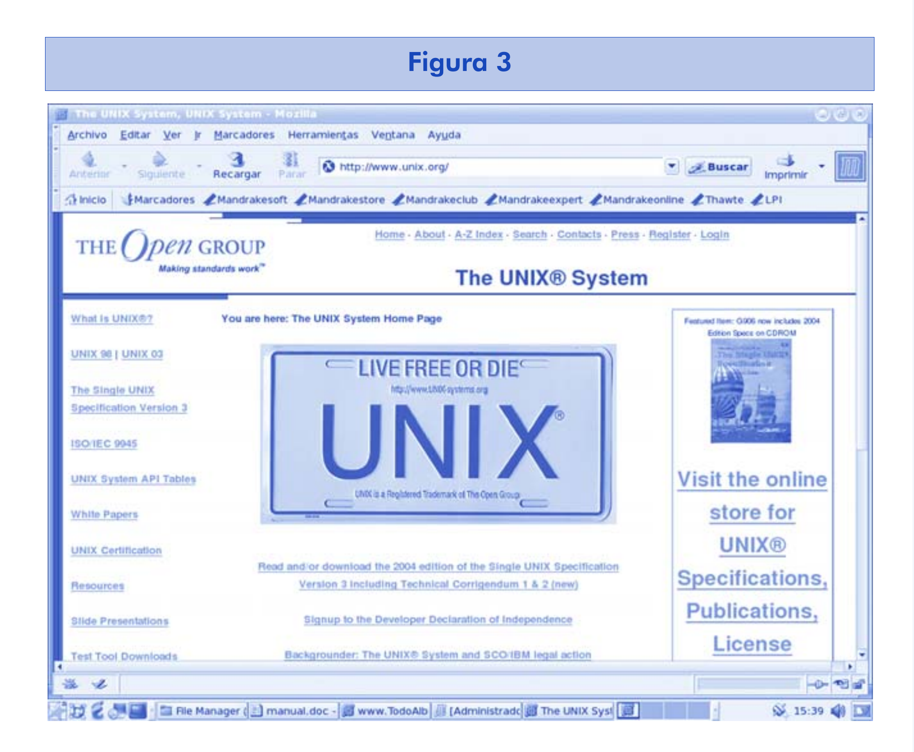
Este producto, así como la página, pertenecen a una empresa denominada Open Group, que está especializada en el desarrollo de estándares globales e integración de información para una interacción global. La tecnología que vende pretende ser neutral y, por tanto, sigue los paradigmas que se persiguen dentro del software libre y en consecuencia del sistema operativo Linux.

UNIX es el sistema operativo predecesor de Linux. Muchas de las características de Linux se heredan de UNIX (sistema robusto, estable, multiusuario, multitarea, multiplataforma, etc.). Por ello esta página proporciona información de gran interés dentro del entorno del software libre. Desde estándar, certificaciones hasta consultas y noticias relacionadas con el sistema operativo. Además posee foros con un objetivo claro de reunir a profesionales para poner

ANOTACIONES

en común conocimientos relacionados con el sistema y el software que se ejecuta en él.

Figura 3



8.3.4. http://www.opensource.org

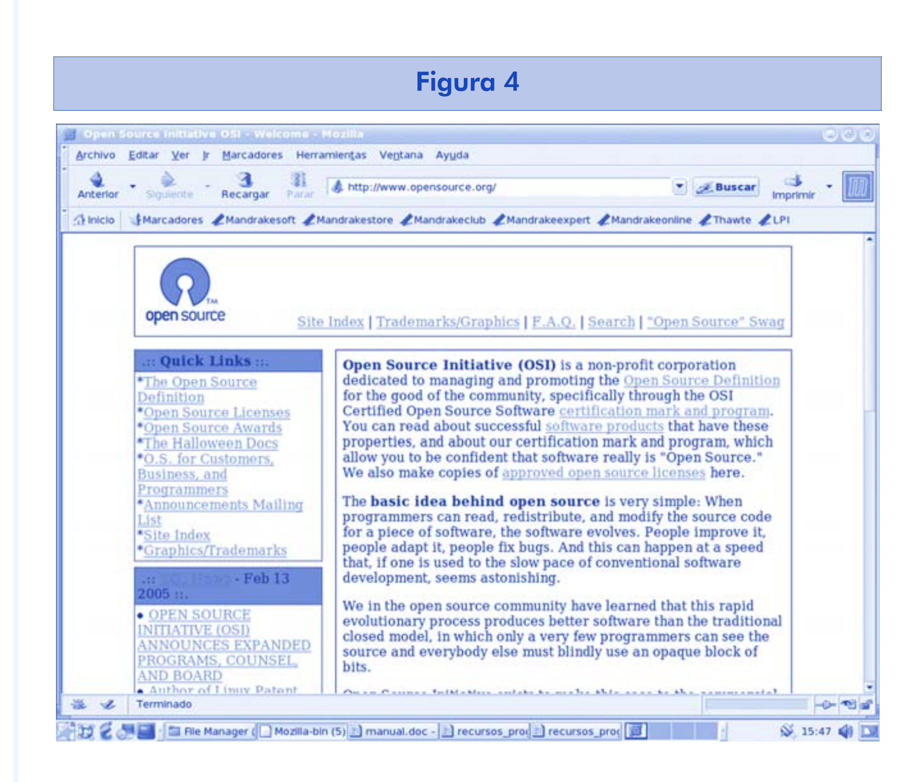
Es una organización sin ánimo de lucro cuyo objetivo es gestionar y promocionar la definición de “código abierto” para el bien de la comunidad.

La licencia “código abierto” implica una serie de características concretas:

1) Libre redistribución.
2) Acceso al código fuente.
3) Permiso de realizar trabajos derivados.
4) Garantía de la integridad del código fuente del autor.
5) No discriminación a personas o grupos.
6) No discriminación a ningún posible campo de la aplicación.
7) Libertad para la distribución de la licencia.
8) La licencia no debe ser específica de un producto.
9) La licencia no debe poner restricciones de uso sobre otro software.

ANOTACIONES

Figura 4



La página ofrece los parámetros, marca las pautas de la definición y define estándares del término "Open Source" y lo que conlleva.

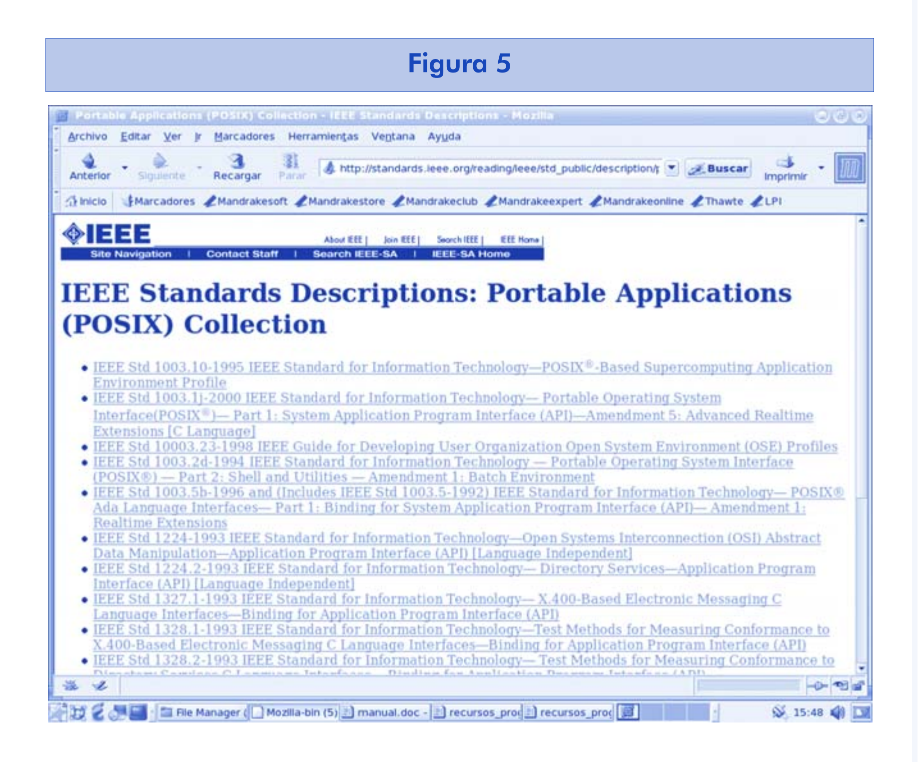
8.3.5. http://standards.ieee.org/reading/ieee/std_public/description/posix/

Esta página se dedica por completo a la definición de estándares bajo la normativa IEEE (Institute of Electrical and Electronic Engineering). Concretamente de estándar POSIX.

POSIX son las iniciales de *portable operating system interface*. Es un estándar que pretende conseguir la portabilidad del software al nivel del código fuente. En otras palabras: un programa escrito para un SO que sea compatible con POSIX ha de poderse compilar y ejecutar sobre cualquier otro "POSIX", aunque sea de otro fabricante distinto. El estándar POSIX define la interfaz que el SO debe ofrecer a las aplicaciones: el juego de llamadas al sistema. POSIX está siendo desarrollado por IEEE y estandarizado por ANSI (American National Standards Institute) e ISO (International Standards Organization). Evidentemente, POSIX está basado en UNIX.

ANOTACIONES

Figura 5



8.3.6. http://www.faqs.org

FAQS (*frequently asked questions* - preguntas frecuentes) pretende responder a todas aquellas preguntas relacionadas con el mundo de la informática, el software, los sistemas operativos, así como otros campos no directamente relacionados con la tecnología.

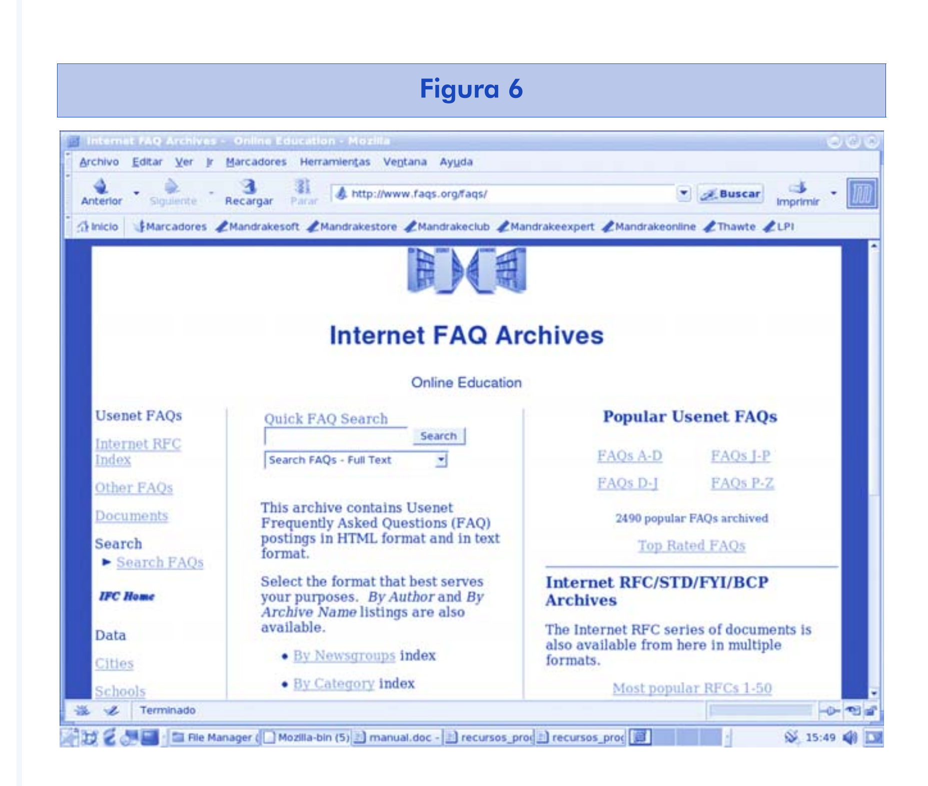
Se puede acceder a ellos por categorías, por autor, por número de referencia, etc.

En cada FAQ tendremos LOS RFC (*request for comments*) correspondientes, es decir, respuestas a las preguntas frecuentes. La base de datos es muy extensa, por lo que muchas veces es necesario hacer búsquedas por alguno de los campos que definen las preguntas y las respuestas.

ANOTACIONES

ANOTACIONES

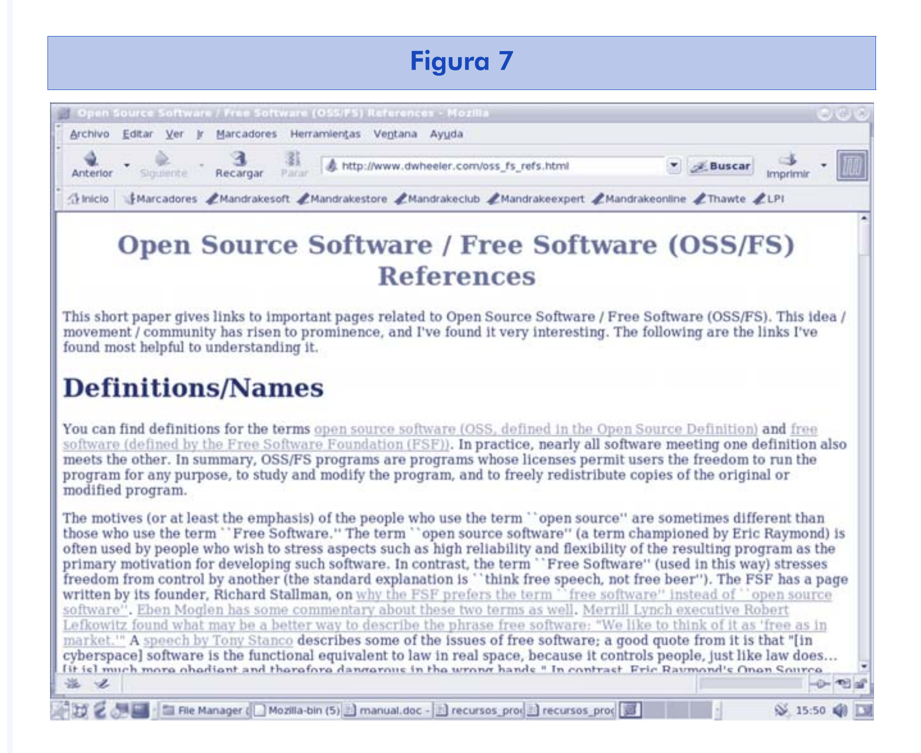
Figura 6



8.3.7. http://www.dwheeler.com/oss_fs_refs.html

Página que ofrece enlaces a páginas relacionadas con el software Open Source y el Free Software(OSS/FS).

Figura 7



Como podemos ver, los recursos documentales son muchos. El ingeniero en SL debe saber, al menos, dónde encontrar documentación sobre las principales aplicaciones que integran los proyectos de software libre, para lo cual lo más cómodo es dirigirse al Linux Documention Project y más concretamente a su zona traducida al español, además de conocer aquellas que le pueden ayudar a resolver problemas de legislación o asimilación de estándares, así como webs de propósito general y de preguntas más corrientes.

8.4. Comunidad

Con la presentación de éstos, intentamos dar únicamente una referencia a la gran cantidad de foros sobre distintos temas relacionados con el software libre. Las webs que detallamos son, en el caso de las relacionadas con preguntas y respuestas, recopilaciones de listas de distribución de gran cantidad de temas. Por otro lado, mostramos una, en la que lo que se publicita es el propio desarrollador, donde puede relacionarse con otros desarrolladores ampliando su red de relaciones.

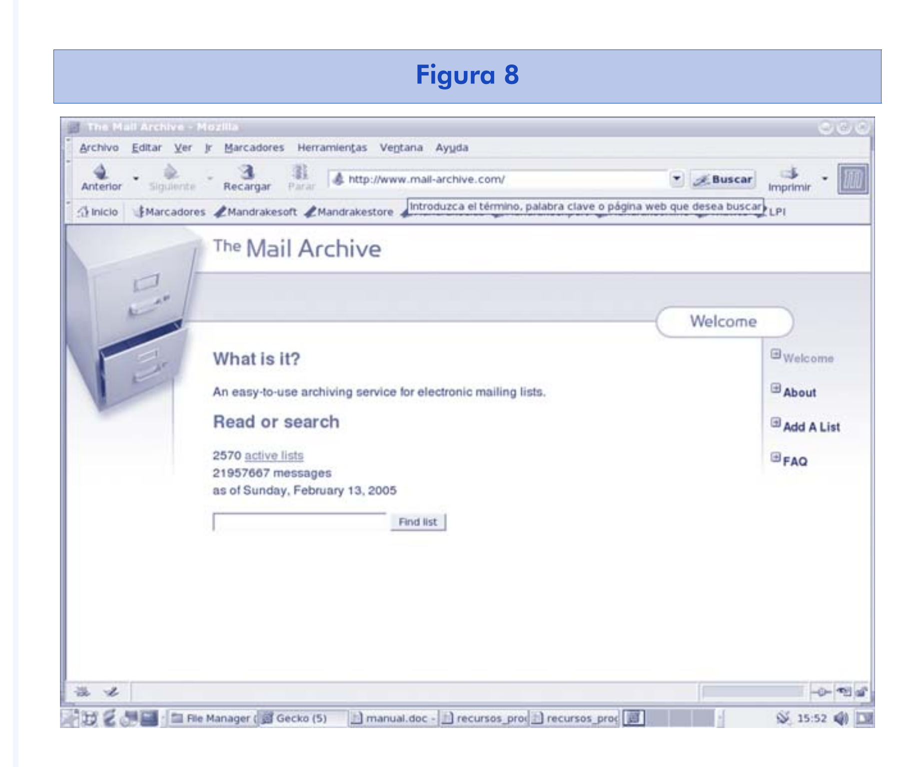
8.4.1. http://www.mail-archive.com

Esta página proporciona un servicio de acceso a archivos pertenecientes a listas de correos públicas. Además, permite añadir nuevas listas para incluir sus archivos para futuras búsquedas.

Es un buen lugar para buscar información de todo tipo, incluida la relacionada con el software libre. La comunidad Linux dispone de una gran cantidad de listas de correo, donde se resuelven dudas y se ponen en común los desarrollos y conocimientos de los usuarios que pertenecen a ellas.

ANOTACIONES

Figura 8

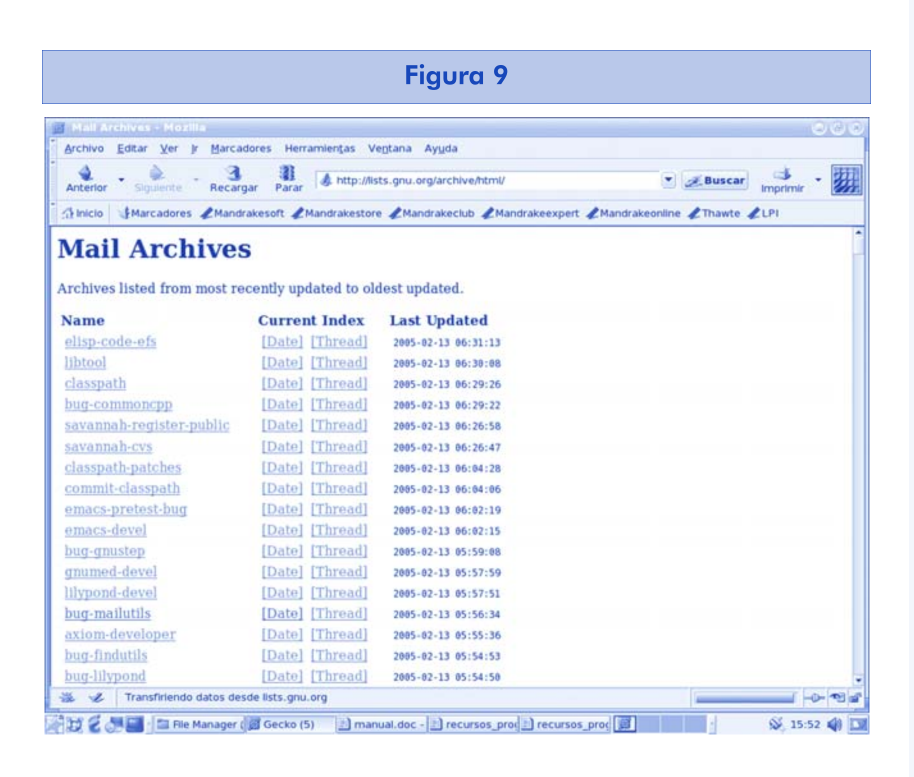


8.4.2. http://mail.gnu.org/archive/html/

La página proporciona todas las listas de correo que pertenecen a GNU. GNU (GNU isn't UNIX) es un proyecto de FSF (Free Software Fundation). Su objetivo es desarrollar un sistema operativo basado en UNIX pero con la idea del software libre de fondo. GNU es el estándar que siguen las nuevas distribuciones Linux para permitir una integración máxima dentro de los sistemas operativos existentes bajo el epígrafe "free software".

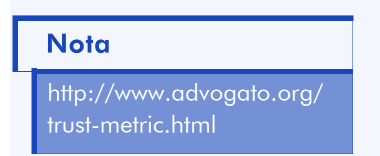
La información a la que accedemos desde esta página nos ayudará a entender mejor la filosofía GNU y además nos servirá de apoyo en el caso de intentar hacer desarrollo de software libre.

Figura 9



8.4.3. http://www.advogato.org/

Advogato, según su *"mission statement"* es un sitio para la comunidad de desarrolladores Open Source. El sitio, con un diseño muy simple, ofrece la posibilidad de crear una cuenta a cualquier persona y dispone de un interesantísimo sistema de certificación intrausuarios llamado trust-metric, que permite que los usuarios del sitio "certifiquen" a otros usuarios. Básicamente una evolución web-style del viejo tema de la validación de usuarios en los BBS.


Nota

http://www.advogato.org/trust-metric.html

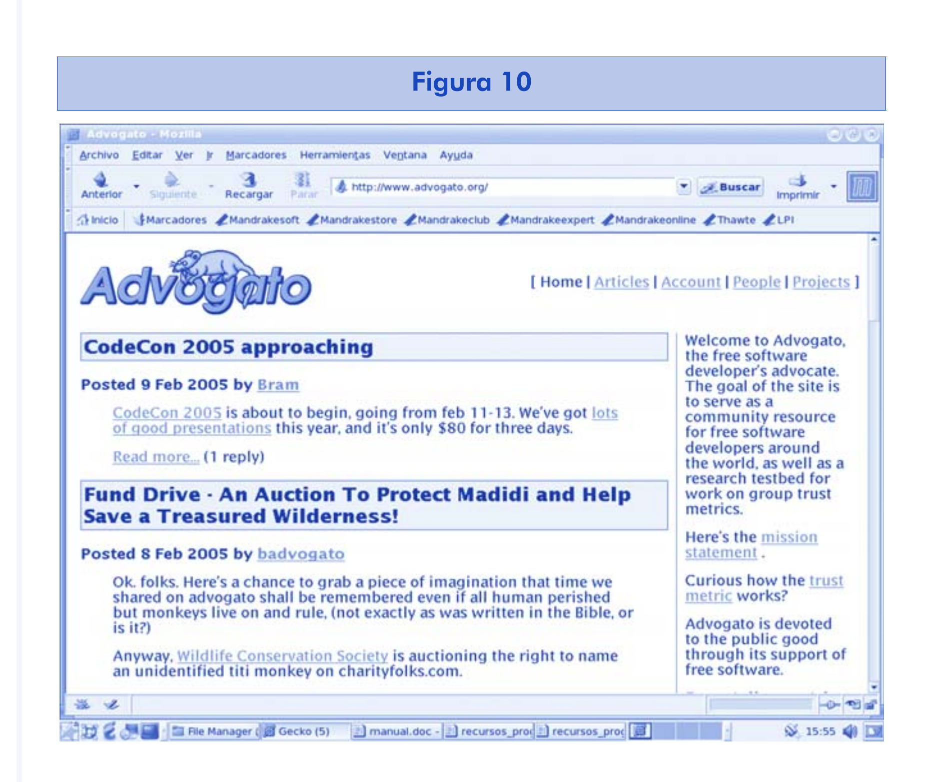
La opción más interesante del sitio consiste en crear un "diario" público que otros pueden leer y comentar, donde los desarrolladores pueden compartir su día a día. Es interesante leer cómo trabajan los demás o qué piensan, y sirve, eventualmente, como una base de lo que uno ha ido haciendo a lo largo del tiempo.

Otras opciones son crear "proyectos", establecer relaciones entre proyectos y usuarios, certificar usuarios y postear artículos (se requiere un cierto nivel de certificación).

El diseño simple y sobrio y la seriedad general del sitio lo hacen más que recomendable. El software con el que está hecho el sitio está disponible, es un modulo en "C" para Apache, algo realmente curioso.

ANOTACIONES

Figura 10



8.5. Albergar proyectos de software libre (sourceforce)

Con este apartado se intenta, a través de Sourceforge –el espacio de albergado y gestión de proyectos más importante del momento–, mostrar la forma como habitualmente se crea una cuenta de proyecto, además de las opciones y facilidades que nos ofrece este sitio.

La operativa, tanto de creación de proyecto como las utilidades que nos dan, son similares. La única barrera puede ser el idioma, para la que presentamos la alternativa hispanohablante, software-libre.org.

8.5.1. Sourceforge.net

Sourcefoge.net es la forja por excelencia. En él se crean y albergan más del 80% de los proyectos de software libre existentes.

Aunque como su nombre indica es una incubadora de proyectos, éstos casi nunca desaparecen de Sourceforge al alcanzar la mayoría de edad, sino que siguen conservando las páginas iniciales del proyecto, aunque tengan su propio dominio y *homepage* en el futuro.

Su principal problema, aunque para cualquier programador que se precie no debería serlo, es el idioma. Su idioma principal es el inglés, aunque podamos albergar un proyecto en español u otro, las herramientas, utilidades y navegación del portal están en inglés.

En este capítulo repasaremos la totalidad de Sourceforge. Crearemos un usuario y un proyecto y una vez creado, daremos un paseo por las utilidades que Sourceforge pone a disposición del programador.

8.5.2. Creación de una cuenta de usuario

La creación de una cuenta de usuario de Sourceforge es sencilla, en primera instancia sólo se nos pide una contraseña y una dirección de correo electrónico.

En un segundo paso debemos introducir un nombre de usuario para el acceso al portal y un nombre publicitario o nombre con el que aparecerá nuestro usuario y los proyectos que alberguemos bajo este nombre.

Además, se nos pregunta el idioma del proyecto y si deseamos recibir correo de las listas de actualización y del resto de comunidades.

Tras realizar el registro completo, recibimos en nuestro correo un mail de confirmación que nos indica que, si queremos terminar nuestro registro, pulsemos sobre el enlace que aparece bajo el epígrafe "[...] *please access the following URL:*"

Al mismo tiempo que recibimos el correo de confirmación, termina el proceso de registro con una página en la que se nos detallan los datos de nuestro usuario: nombre de la cuenta de usuario, alias para la cuenta de creación, por ejemplo, usuario@dominio.com, cuenta de Sourceforge, la cual es un alias a nuestra cuenta real, la página en la que podemos modificar nuestros datos de usuario y en la que podemos encontrar información, documentación y respuestas a nuestras preguntas más corrientes.

Una vez creada la cuenta, podemos acceder a ella con nuestro nombre de usuario y nuestra contraseña. Esta última la introdujimos en

ANOTACIONES

el primer paso de creación de la cuenta. Además, desde esta misma podemos solicitar nueva contraseña en caso de olvido o crear una nueva cuenta.

En el momento del acceso se abre nuestra página de usuario, donde aparece toda la información del mismo, si tenemos o no proyectos en curso o pendientes de activación, si se han realizado donativos a nuestro usuario y si tenemos alguna pregunta pendiente con el centro de apoyo entre otras.

8.5.3. Alta de nuevo proyecto

La primera vez que creamos una cuenta, no se nos pide que introduzcamos un nombre real para poder activarla, pero este proceso sí que es necesario en el caso de querer dar de alta un proyecto nuevo.

Por otra parte, y dependiendo del tipo de usuario que hemos creado y del tipo de licencia que utilizamos, puede ser interesante que habilitemos los donativos; para hacerlo, debemos disponer de cuenta en PayPal. Este sistema permite transferir fondos a una cuenta de usuario sin necesidad de utilizar tarjeta de crédito.

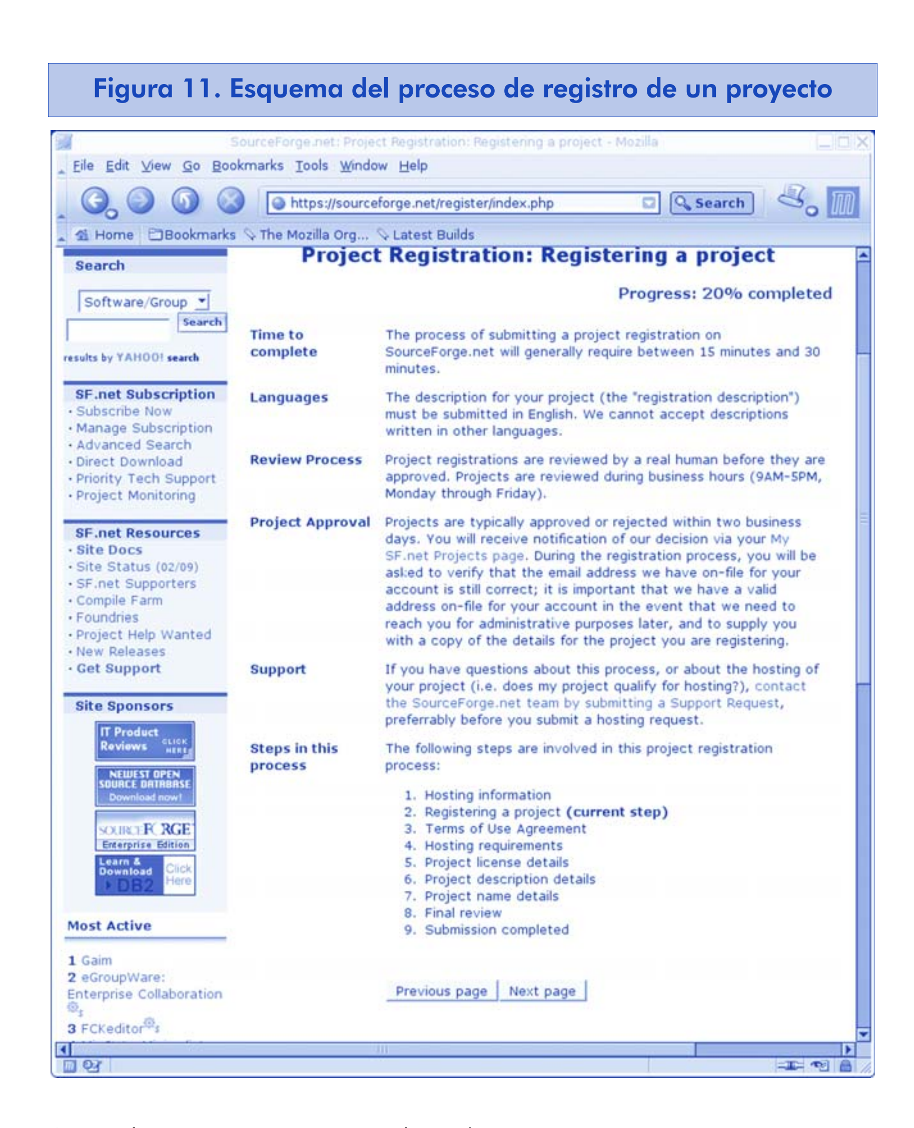
Cualquiera que juzgue útil nuestro proyecto podrá donar la cantidad de dinero que crea oportuna y ayudarnos así a mantenerlo.

En la creación de un nuevo proyecto, aunque no es necesario que éste sea libre, se nos indica, por si no lo sabemos, qué ventajas tiene el SL y la creación de este tipo de proyectos frente al software propietario.

El primer paso es indicar el tipo de proyecto que queremos publicar, en nuestro caso, documentación. Por otro lado, se nos pregunta si hemos entendido el significado de la definición de Open Source.

ANOTACIONES

Figura 11. Esquema del proceso de registro de un proyecto



Antes de continuar, se nos indican los pasos que vamos a seguir y en primer lugar el tiempo que nos tomará este proceso de alta de proyecto.

Aceptamos los términos y condiciones para albergar un proyecto en Sourceforge e incluimos los datos de la licencia. Si nuestro proyecto se desarrolla bajo una licencia que no aparece en el listado, por ejemplo, GFDL, escribimos una breve descripción de la misma en el recuadro inferior.

El siguiente paso es escribir una descripción pública y otra para su registro en la base de datos de Sourceforge.

Por último se nos pide que repasemos todos los datos introducidos por si hubiese algún error.

Tras este repaso, recibimos un correo que confirma la creación del proyecto y el paso a aprobación, es decir, el proyecto será revisado

ANOTACIONES

por personal de Sourceforge y albergado definitivamente. Este proceso suele durar dos o tres días hábiles.

8.5.4. Utilidades que Sourceforge pone a disposición de los usuarios de un proyecto

Una vez terminado el proceso de creación de un proyecto, se ponen a disposición del administrador del proyecto una serie de herramientas que permitirán el correcto trabajo colaborativo de los diferentes partícipes del mismo:

En su barra de herramientas podemos hacernos una idea de lo que se nos ofrece.

Figura 12. Barra de herramientas de proyecto

Project: Gaim: Summary

Summary | Admin | Home Page | Forums | Tracker | Bugs | Support | Patches | RFE | Lists | Screenshots | News | CVS | Files |

En este caso es para el proyecto Gaim, que proporciona el desarrollo de un cliente de mensajería instantánea compatible con casi todos los servidores disponibles.

Como vemos, se nos muestra un sumario del proyecto con su descripción, áreas públicas disponibles y últimas noticias.

Figura 13. Descripción del proyecto

Gaim is a GTK2-based instant messenger application. It supports multiple protocols via modules, including AIM, ICQ, Yahoo!, MSN, Jabber, IRC, Napster, Gadu-Gadu and Zephyr. It has many common features found in other clients, as well as many unique featur

Foundry Member :GNOME Foundry

Project of the month for: October 2002

- Development Status: 5 - Production/Stable
- Intended Audience: End Users/Desktop
- License: GNU General Public License (GPL)
- Operating System: All 32-bit MS Windows (95/98/NT/2000/XP), All BSD Platforms (FreeBSD/NetBSD/OpenBSD/Apple Mac OS X), All POSIX (Linux/BSD/UNIX-like OSes), Linux, Other
- Programming Language: C
- Topic: AOL Instant Messenger, ICQ, Internet Relay Chat
- User Interface: Gnome, Win32 (MS Windows)

Project UNIX name: gaim
Registered: 1999-11-13 23:51

Activity Percentile (last week): 100%

Arriba podemos ver la descripción del proyecto. Y en las siguientes, las áreas públicas disponibles.

ANOTACIONES

Figura 14. Áreas públicas disponibles

Public Areas

Project Home Page

Tracker

- Bugs (**605 open / 9035 total**)
Bug Tracking System

- Support Requests (**64 open / 662 total**)
Tech Support Tracking System

- Patches (**69 open / 1920 total**)
Patch Tracking System

- Feature Requests (**1532 open / 2524 total**)
Feature Request Tracking System

- Plugins (**61 open / 266 total**)
Plugin Tracking System

- gtk-bugs (**112 open / 147 total**)
Gtk only wants reports with non-gaim test cases.

- Rejected Patches (**3 open / 4 total**)
Patches not accepted to the Gaim core, but that might be useful to someone.

- Translations (**8 open / 10 total**)
A tracker for updated .po files

Public Forums (**12177** messages in **1** forums)

Mailing Lists (**9 total**)

Screenshots

CVS Repository (**13,529** commits, **3,435** adds)
- Browse CVS

Y también las últimas noticias:

Aparecen listadas por fecha las últimas versiones y adiciones al proyecto.

ANOTACIONES

Figura 15. Últimas noticias

Latest News

Gaim 0.60 released (Linux and win32)
robflynn - 2003-04-05 18:01
(11 Comments) [Read More/Comment]

Gaim v0.60-alpha3 for win32
robflynn - 2002-11-08 15:46
(5 Comments) [Read More/Comment]

Gaim v0.59.6 released
robflynn - 2002-11-08 15:43
[Read More/Comment]

Gaim v0.54 Release; Direct IM Image Support
robflynn - 2002-03-15 21:43
(10 Comments) [Read More/Comment]

Gaim v0.49 released!
robflynn - 2001-11-29 22:19
[Read More/Comment]

Applet RPMS available
robflynn - 2001-06-05 05:49
(2 Comments) [Read More/Comment]

GAIM 0.11.0pre5 Has been released!
robflynn - 2001-02-27 00:37
(1 Comment) [Read More/Comment]

0.11.0pre2
robflynn - 2000-12-04 22:20
[Read More/Comment]

0.11.0pre1
robflynn - 2000-12-04 08:16
(1 Comment) [Read More/Comment]

0.10.2 - wahoo
robflynn - 2000-10-07 21:44
[Read More/Comment]

[News archive]
[Submit News]

Zona de descargas de últimas versiones:

Tanto de versiones finales como de utilidades o software adicional necesario.

Figura 16. Zona de descargas

Latest File Releases

Package	Version	Date	Notes / Monitor	Download
gaim	1.1.2	January 21, 2005	-	Download
GTK+ for Windows	2.4.14 Rev A	January 21, 2005	-	Download

ANOTACIONES

Zona de foros

Figura 17. Foros



Seguimiento de áreas de trabajo, así como zona de errores y parches. Todos ellos incluidos como opciones visibles de un completo sistema de control de versiones.

Por lo tanto, en Sourceforge no sólo incluimos nuestro proyecto, sino que nos dan el espacio necesario para albergarlo y multitud de herramientas que ayudan a los programadores a colaborar. Comunicándose con los foros, enviando sus trabajos con el CVS, además de pequeñas utilidades como la redirección web para nuestro proyecto o la redirección del correo electrónico de nuestra cuenta de Sourceforge a nuestra cuenta personal.

Todo esto sólo bajo la recomendación de desarrollar nuestro proyecto como un proyecto abierto donde se permita la colaboración de otros programadores y bajo un sinfín de licencias abiertas.

Desde hace relativamente poco se ha añadido a Sourceforge la opción power, en la que por un precio asequible se ofrecen al usuario ciertos servicios no disponibles en la gratuita como:

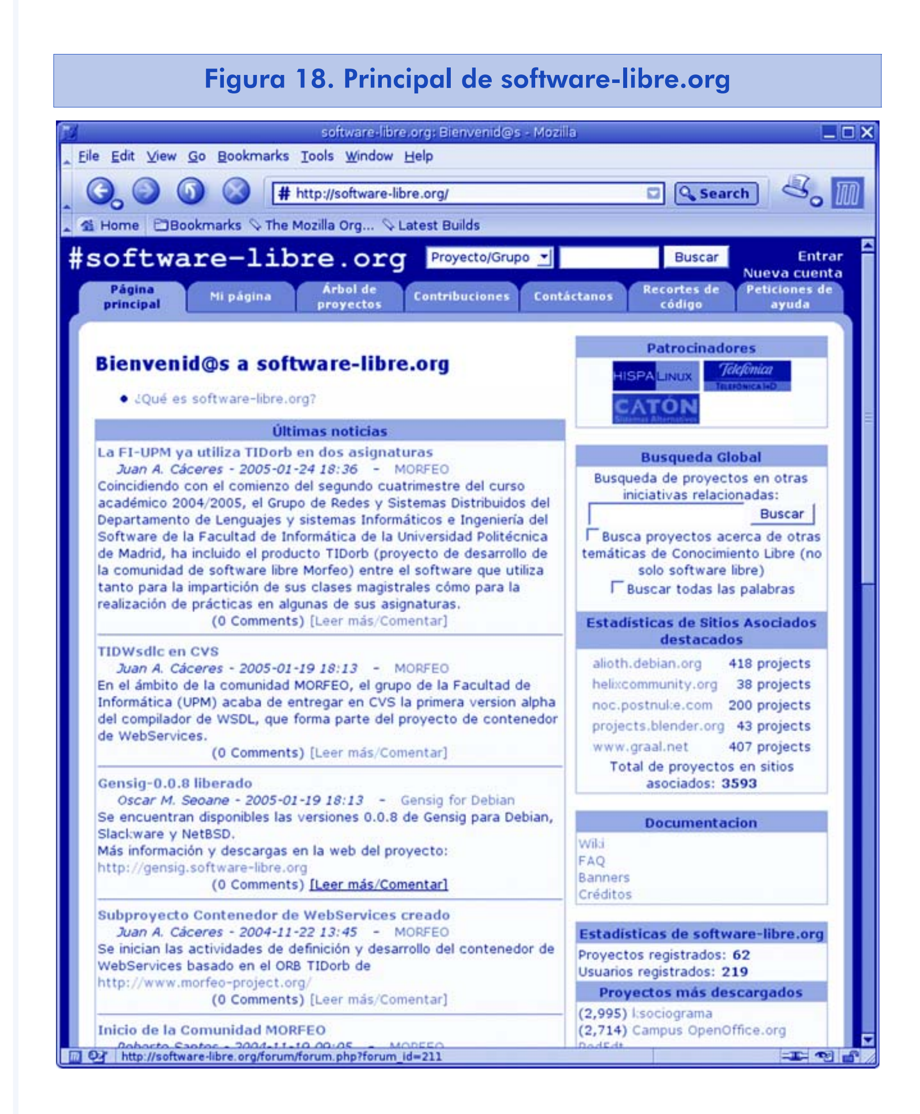
- Acceso a utilidades de búsqueda avanzada para los proyectos de Sourceforge
- Soporte técnico prioritario
- Seguimiento de proyectos
- Descargas directas de software sin pasar por servidores de réplica

8.5.5. Software-libre.org

El homónimo español de Sourceforge es software-libre.org. Este portal utiliza el mismo software que Sourceforge y alberga proyectos gratuitamente pero en español.

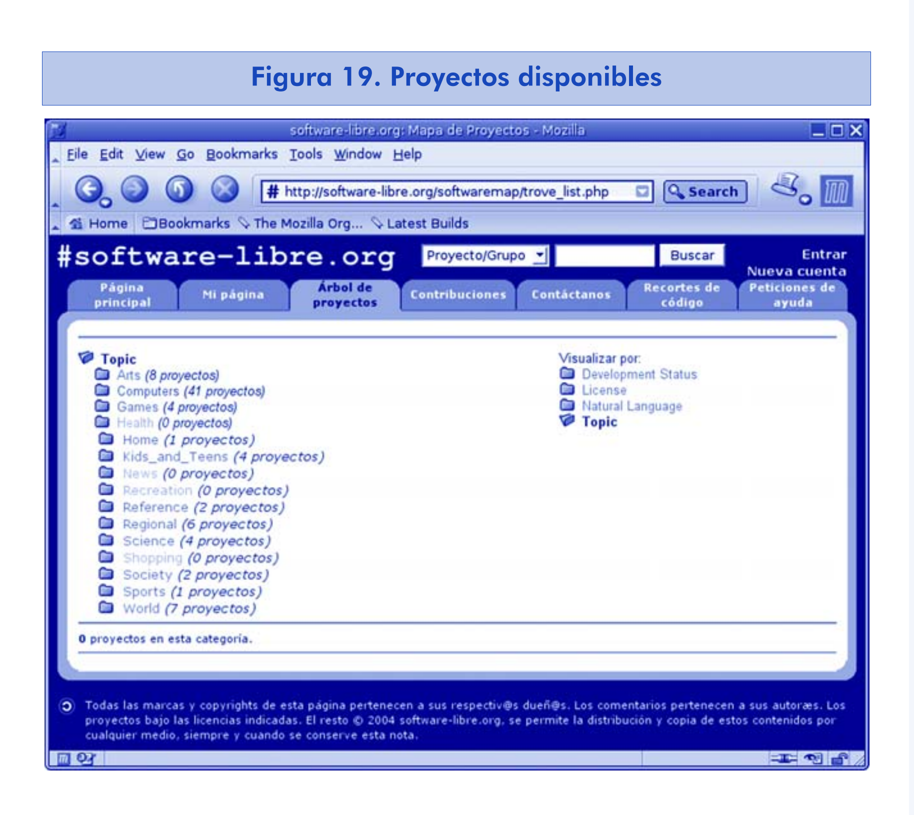
Es una opción muy válida para todos aquellos desarrolladores hispanos que quieran publicitar proyectos en su zona de influencia o que no conozcan el idioma inglés.

Figura 18. Principal de software-libre.org



Su lista de proyectos es mucho menor que la de Sourceforge y también es mucho más joven. Nació en el 2003, por lo que su desarrollo ha sido mucho menor.

Figura 19. Proyectos disponibles



El proceso de registro de usuarios y proyectos es muy similar al de Sourceforge.

Ésta puede ser una buena opción, aunque muy localizada, para albergar nuestros proyectos. De momento su uso es limitado y casi localizado en España, pero el sistema está creciendo continuamente. Esperemos que en poco tiempo pueda ser referencia hispana como forja de proyectos libres.

8.5.6. Publicitar proyectos de software libre y obtener notoriedad

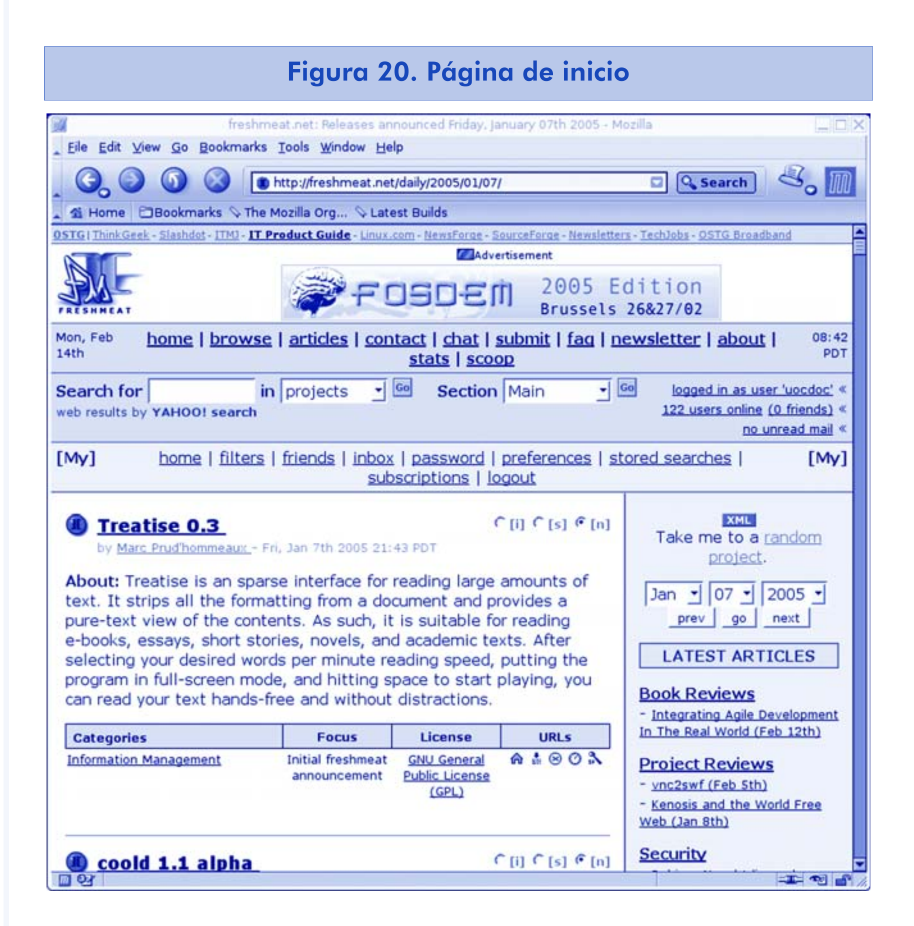
Cada día más, es necesario –tanto para encontrar colaboradores como patrocinadores– publicitar nuestros proyectos con la herramientas adecuadas. Casi ninguna cumple todas las expectativas, pero sí que podemos lograr, usando varias de ellas, el objetivo deseado.

La más usada de todas para publicitar proyectos de software y relacionarse con otros desarrolladores es Freshmeat.net.

Vamos a introducirnos en esta herramienta para más adelante mostrar otra de apoyo que completará nuestro objetivo.

8.5.7. Freshmeat.net

Figura 20. Página de inicio



Como podemos ver en su página principal, con Freshmeat conseguiremos publicitar nuestro proyecto software haciendo que éste aparezca en la lista de su página principal, de forma que tanto desarrolladores interesados en participar, como usuarios interesados en usarlo puedan acceder a él sin necesidad de buscarlo.

Proporciona un control de versiones que nos permite ir añadiendo sucesivas revisiones y poder comprobar su historial, al mismo tiempo que nos pone en contacto con otros desarrolladores de proyectos parecidos con los que poder establecer cooperaciones.

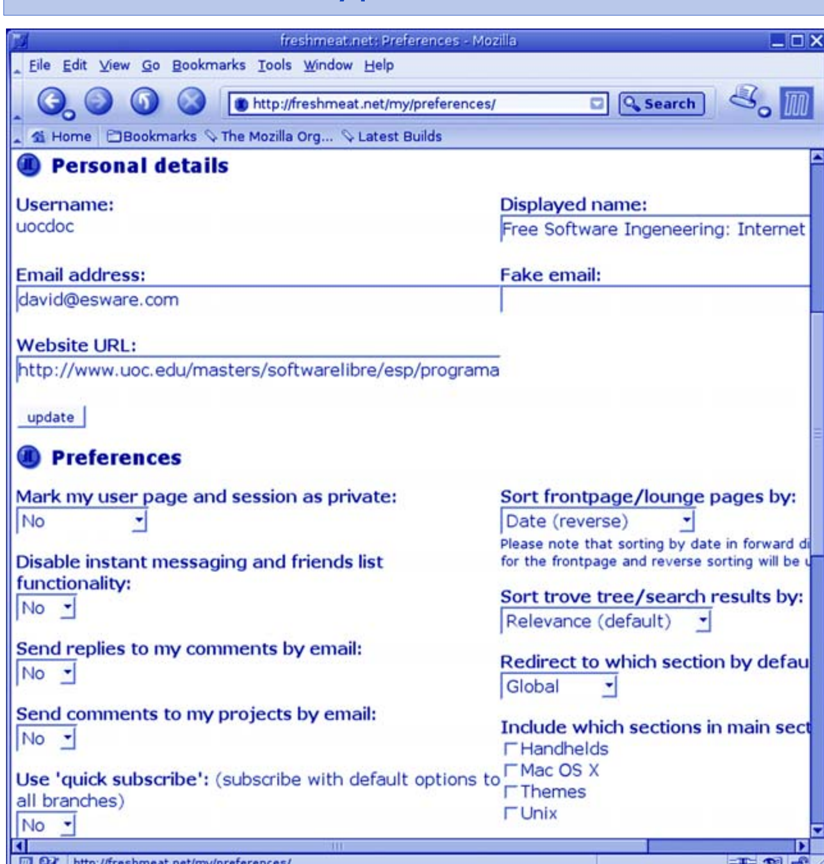
Vamos a crear una cuenta de usuario y a seguir el proceso de creación de un proyecto ficticio:

- La creación de una cuenta de usuario es sencilla. Sólo se nos solicita un nombre de usuario, un nombre de proyecto que será visible en la lista principal de Freshmeat, una dirección de correo y una contraseña. Como dato opcional podemos introducir el sitio web del proyecto.

- Una vez creado el usuario, recibimos un mensaje de confirmación y activación. Como en el caso de Sourceforge, debemos pulsar sobre el enlace para activar nuestra cuenta.

Una vez creado el usuario desde su página principal, podemos realizar las tareas necesarias para la administración de nuestra cuenta y sus preferencias personales:

Figura 21. Administración de preferencias de proyecto y personales



Y también para la creación de un nuevo proyecto de visibilidad. Todas estas opciones podemos realizarlas desde dos menús:

- El principal

Figura 22. Menú principal

home | browse | articles | contact | chat | submit | faq | newsletter | about | stats | scoop

Desde el que podemos añadir un nuevo proyecto o acceder a información general del portal.

ANOTACIONES

- Y el de usuario

Figura 23. Menú de usuario

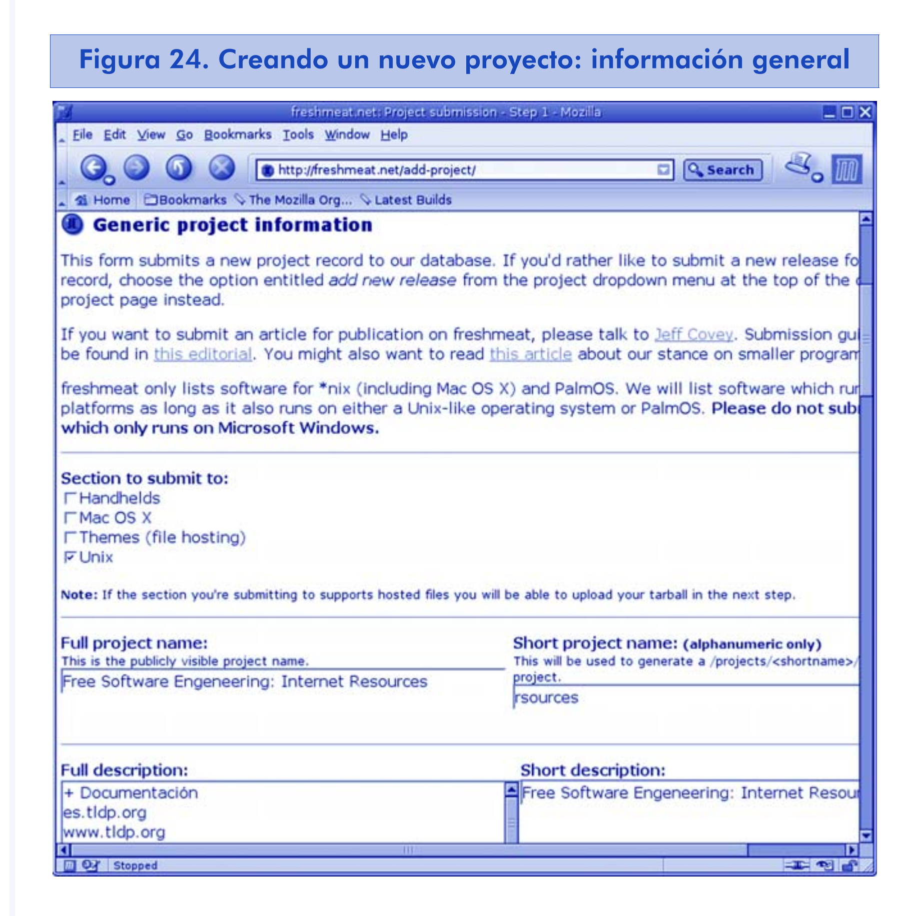
home | filters | friends | inbox | password | preferences | stored searches | subscriptions | logout

Desde donde podemos añadir filtros para ver sólo aquellos proyectos que nos interesen, acceder a los mensajes que nos puedan haber enviado otros usuarios, a las preferencias de nuestro usuario, a las estadísticas, etc.

La opción de amigos es realmente interesante, ya que desde ella mantenemos nuestra lista de contactos de Freshmeat, una de las cosas más importantes a la hora de publicitar nuestro proyecto y e-colaborar con el resto de la comunidad.

¿Como añadir un nuevo proyecto? Sólo tenemos que pulsar en el enlace submit del menú principal y podremos hacerlo:

Figura 24. Creando un nuevo proyecto: información general

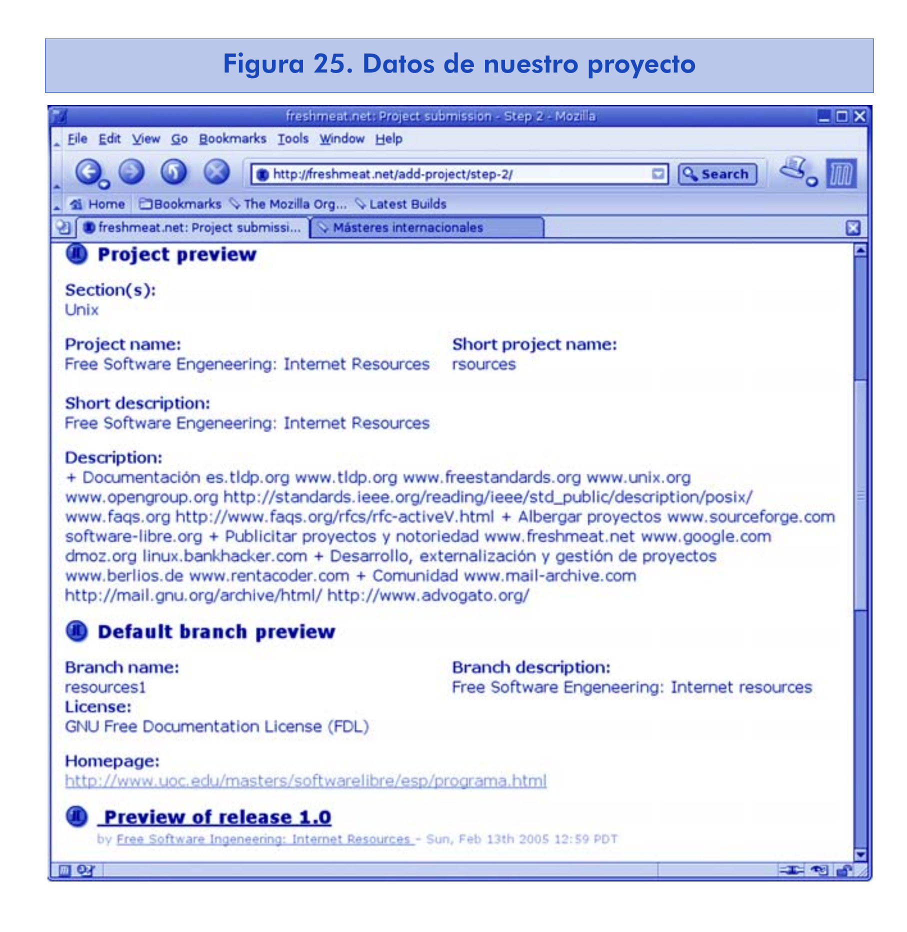


Como vemos es sencillo, sólo debemos escoger el sistema operativo del mismo, un nombre completo de proyecto, su nombre corto, su descripción completa y su descripción corta.

ANOTACIONES

Añadir el proyecto a una categoría y repasar los datos que hemos introducido como muestra la siguiente imagen.

Figura 25. Datos de nuestro proyecto



Con estos tres simples pasos y esperando la confirmación del administrador del sitio, que suele tardar unos dos días, dispondremos de nuestro proyecto creado y listo para mandar comunicaciones y avisos de nuevas versiones o noticias.

Descripción funcional de Freashmeat

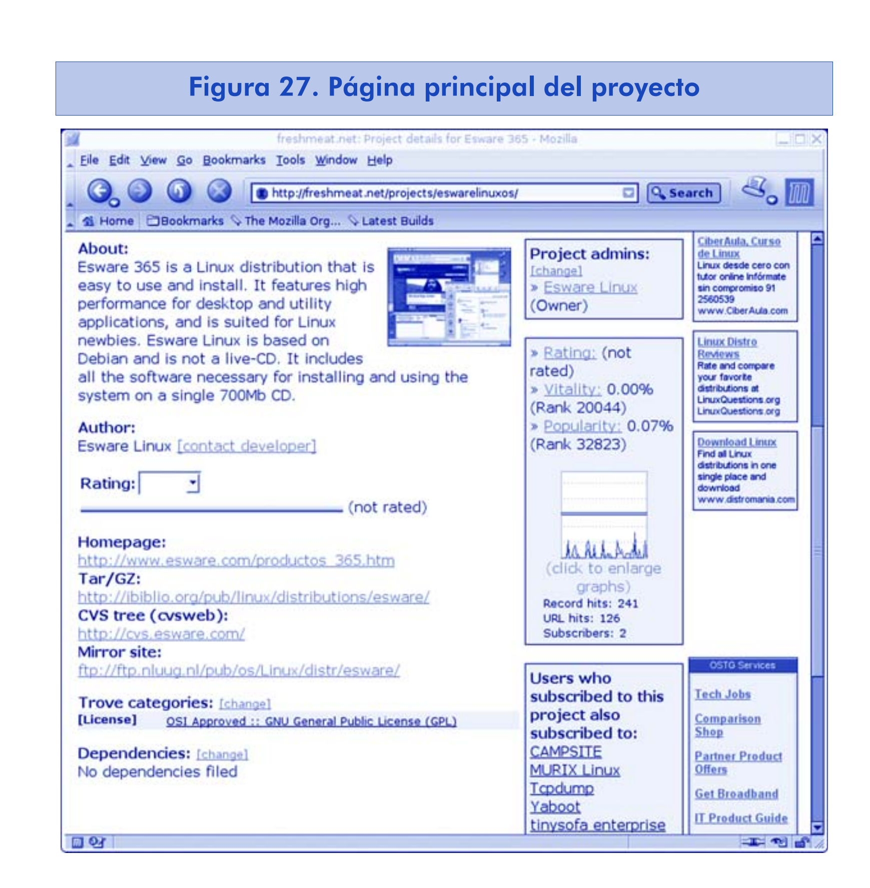
Desde el menú de usuario y con un proyecto dado de alta, podemos acceder a la administración del mismo, mediante el menú de proyecto:

Figura 26. Menú de proyecto

add release | add branch | add screenshot | broken links | change owner
email subscribers | update project | update branch (urls)

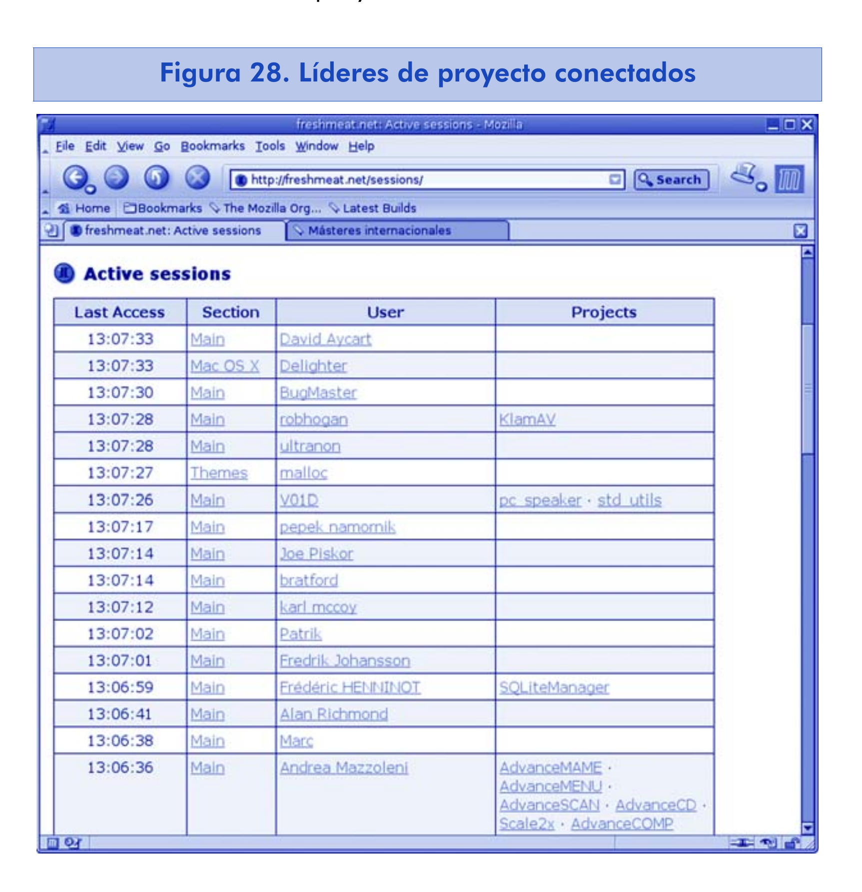
Y a la página principal del proyecto desde la que podemos ver los subscriptores a ese proyecto, las estadísticas de visita sobre el mismo, una descripción y una imagen de previsualización de la herramienta además de páginas de interés sobre la misma:

Figura 27. Página principal del proyecto



Aunque la mejor utilidad ofrecida por Freshmeat es su lista de desarrolladores desde la que se puede localizar a la practica totalidad de los mantenedores de los proyectos de software libre más conocidos.

Figura 28. Líderes de proyecto conectados



Last Access	Section	User	Projects
13:07:33	Main	David Aycart	
13:07:33	Mac OS X	Delighter	
13:07:30	Main	BugMaster	
13:07:28	Main	robhogan	KlamAV
13:07:28	Main	ultranon	
13:07:27	Themes	malloc	
13:07:26	Main	V01D	pc_speaker · std_utils
13:07:17	Main	pepek_namornik	
13:07:14	Main	Joe Piskor	
13:07:14	Main	bratford	
13:07:12	Main	karl mccoy	
13:07:02	Main	Patrik	
13:07:01	Main	Fredrik Johansson	
13:06:59	Main	Frédéric HENNINOT	SQLiteManager
13:06:41	Main	Alan Richmond	
13:06:38	Main	Marc	
13:06:36	Main	Andrea Mazzoleni	AdvanceMAME · AdvanceMENU · AdvanceSCAN · AdvanceCD · Scale2x · AdvanceCOMP

ANOTACIONES

8.6. ¿Cómo obtener notoriedad para nuestros proyectos?

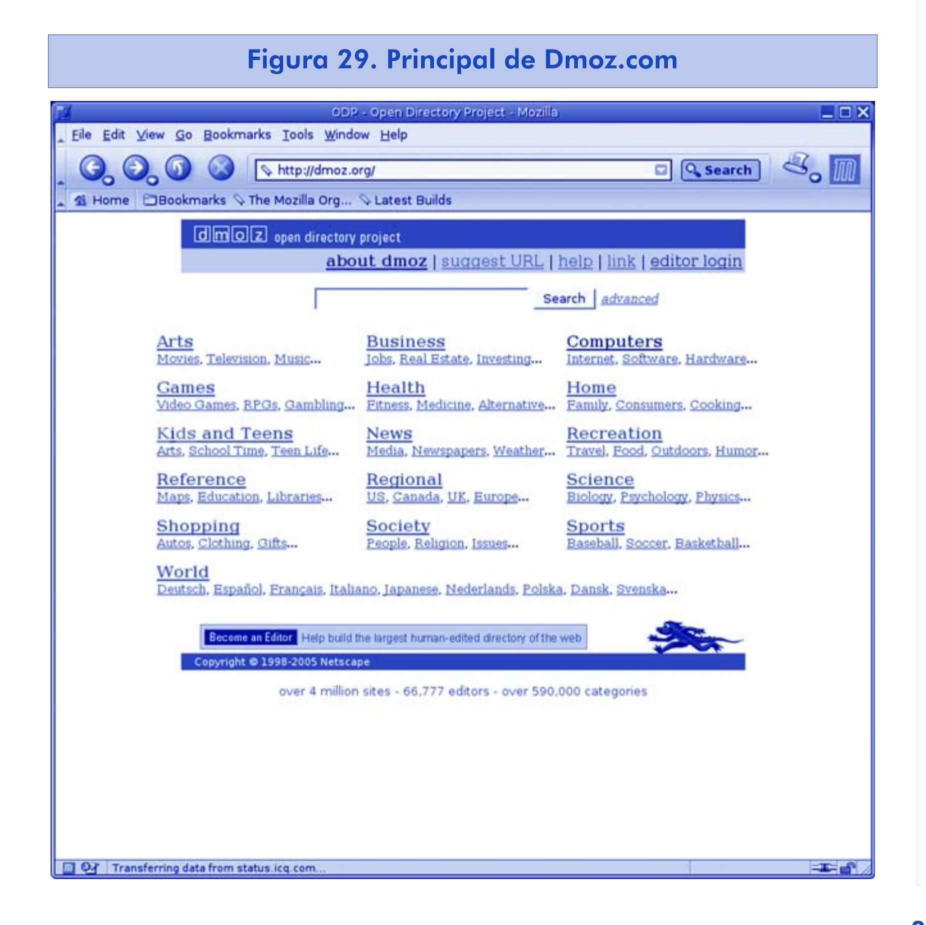
La notoriedad para nuestros proyectos y para nosotros mismos se obtiene por regla general con el trabajo de años, pero también con ayuda de índices temáticos y buscadores.

Vamos a dar como orientación tres de ellos, intentando además que éstos sean referencias hispanas, cosa algo difícil.

- **dmoz.com.** Dmoz es un índice temático que intenta indexar páginas adjuntas a determinados epígrafes, de modo que la búsqueda resulte mucho más natural y centrada.

  En Dmoz participan gran número de editores de contenido que se dedican a mantener páginas de confianza dentro de epígrafes de confianza, es decir, si nosotros sabemos de la existencia de una página, de la que podemos ser los administradores, podemos hacernos editores de la misma en una sección determinada, por ejemplo, desarrollo de software, software libre, etc., de manera que al mantenerla en el índice la publicitamos.

Figura 29. Principal de Dmoz.com



ANOTACIONES

- Contenido de un directorio

Dentro del directorio computers, existen infinidad de subdirectorios entre los que está el de Open Source. Podemos mantener nuestro proyecto en su interior.

Figura 30. Página de recursos sobre informática

- **linux.bankhacker.com.** Una página española bastante consultada para proyectos es la del grupo de desarrolladores de bankhacker. En ella se pueden sugerir productos de software para incluirlos en el directorio de software libre y Linux.

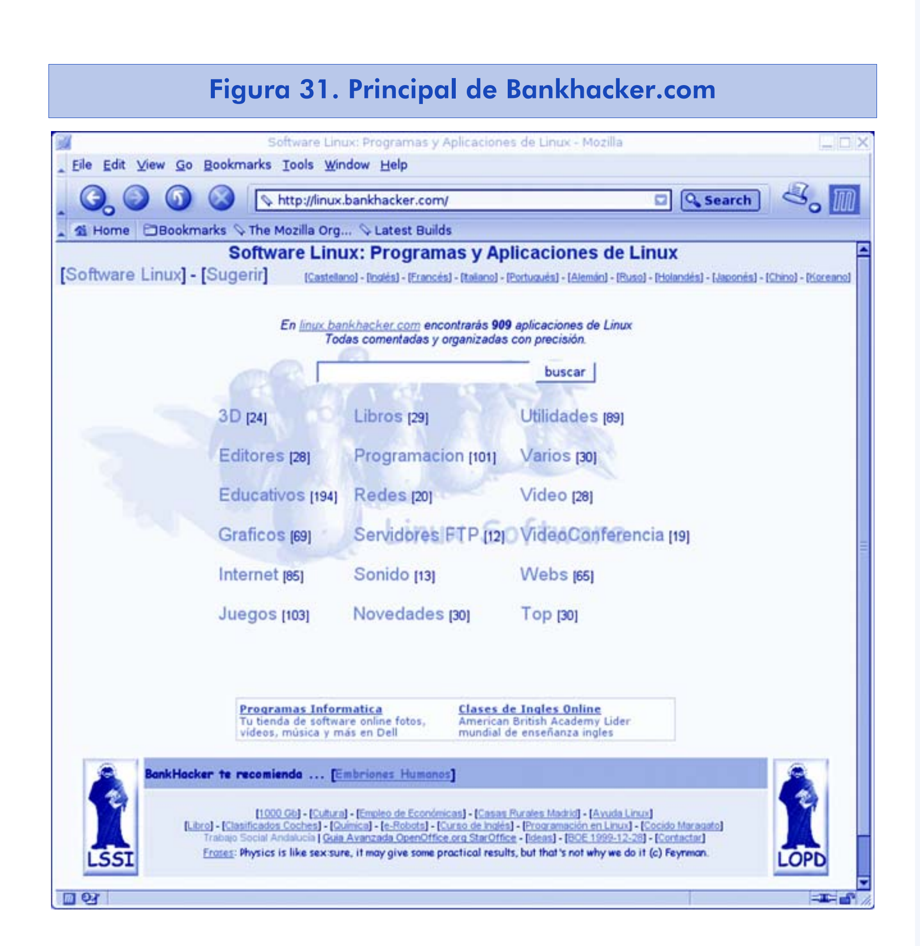


Figura 31. Principal de Bankhacker.com

Podemos sugerir una aplicación en concreto rellenando el formulario que aparece tras pulsar el enlace "sugerir".

8.7. Conclusiones

La conclusión final de este apartado está clara. Disponemos de multitud de recursos útiles para los ingenieros del software libre. No hay uno solo para albergar proyectos, pero sí, uno más importante que los demás. Lo mismo ocurre con los recursos para publicitar nuestros proyectos y a nosotros mismos como desarrolladores, hay uno más importante, como es Freshmeat, pero no es el único, sino que existen otros que cubren deficiencias de éste.

Lo que está claro es que cada día salen nuevos recursos útiles y nuevos proyectos recomendables, que no siempre se publicitan en los mismos sitios y que no siempre llegamos a conocerlos.

Se necesita una centralización a la que se está llegando mediante la sindicación de contenidos y noticias, pero aún quedan barreras por superar y una de las más patentes es la del idioma.

Hoy por hoy, los desarrolladores deben conocer el inglés para poder dar a conocer sus proyectos. Esperemos que en un futuro próximo los recursos principales también estén traducidos para acercarlos a los desarrolladores no angloparlantes.

ANOTACIONES

Appendix A. GNU Free Documentation License

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of *copyleft* which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

ANOTACIONES

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The *Document* below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as *you* You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A *Modified Version* of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A *Secondary Section* is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The *Invariant Sections* are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The *Cover Texts* are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

ANOTACIONES

A *Transparent* copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not *Transparent* is called *Opaque*.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The *Title Page* means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, *Title Page* means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section *Entitled XYZ* means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as *Acknowledgements*, *Dedications*, *Endorsements*, or *History*. To *Preserve the Title* of such a section when you modify the Document means that it remains a section *Entitled XYZ* according to this definition.

ANOTACIONES

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

ANOTACIONES

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the

ANOTACIONES

Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled *History*. Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled *History* in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the *History* section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled *Acknowledgements* or *Dedications* Preserve the Title of the section, and preserve in the section all the

ANOTACIONES

substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- M. Delete any section Entitled *Endorsements* Such a section may not be included in the Modified Version.

- N. Do not retitle any existing section to be Entitled *Endorsements* or to conflict in title with any Invariant Section.

- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled *Endorsements*, provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

ANOTACIONES

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled *History* in the various original documents, forming one section Entitled *History*; likewise combine any sections Entitled *Acknowledgements*, and any sections Entitled *Dedications*. You must delete all sections Entitled *Endorsements*.

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this

ANOTACIONES

License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an *aggregate* if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a

ANOTACIONES

translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled *Acknowledgements*, *Dedications*, or *History*, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License *or any later version* applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version

ANOTACIONES

number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the *with...Texts.* line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

ANOTACIONES


UOC
www.uoc.edu